L'ABC DEL LINGUAGGIO MACCHINA SUL COMMODORE 64

Per chi non si accontenta del BASIC

di DANNY DAVIS

LIBRO + CASSETTA

edizioni JCE

L'ABC DEL LINGUAGGIO MACCHINA SUL COMMODORE 64

Per chi non si accontenta del BASIC

di DANNY DAVIS

Traduzione di A. BARATTINI

JACOPO CASTELFRANCHI EDITORE
Via dei Lavoratori, 124
CINISELLO BALSAMO (MI)

Tutti i diritti sono riservati, nessuna parte di questo libro puo' essere riprodotta, posta in sistemi di archiviazione, trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopiatura, ecc., senza la preventiva autorizzazione scritta dell'editore.

I listati dei programmi presentati in questo libro sono stati scelti tenendo conto del loro valore didattico, controllati con cura, non sono peraltro garantiti per applicazioni particolari. L'editore, malgrado l'accurata realizzazione, non puo' essere chiamato responsabile degli eventuali errori che potrebbero comparire durante l'esecuzione.

Il possessore di questo libro non puo' utilizzare il contenuto e i programmi per impieghi differenti da quello di studio personale, senza la preventiva autorizzazione del proprietario del copyright.

Titolo originale: Machine Language for the Absolute Beginner.
Pubblicato nel Regno Unito
dalla Melbourne House (Publishers) Ltd.,
Pubblicato in Australia
dalla Melbourne House (Australia) Pty. Ltd.
Pubblicato negli Stati Uniti d'America
dalla Melbourne House Sotware Inc.

Copyright © per l'edizione inglese Beam Software, 1984
Copyright © per l'edizione italiana JCE, 1985

Prima edizione: GIUGNO 1985

Stampato in Italia da
Genm Grafica S.r.l.
Via Magretti - Paderno Dugnano (MI)

INDICE

Prefazione 4

CAPITOLO 1

Introduzione al linguaggio macchina 5

Uso di un programma in linguaggio macchina 6
Indirizzamento della memoria 6
Sommario 8

CAPITOLO 2

Basi della programmazione in linguaggio macchina 11

I registri 11
Metodi di indirizzamento 12
Linguaggio simbolico 15
Memoria di schermo 16
Stampa di un messaggio 18
Sommario 20

CAPITOLO 3

Introduzione all'esadecimale 23

Binario 23
Perche' esadecimale? 25
Indirizzamento assoluto 28
Conversione da esadecimale a decimale 29
Sommario 30

CAPITOLO 4

Introduzione all'ALPA 31

Sommario 36

CAPITOLO 5

Equipaggiamento del microprocessore 37

Memorizzazione di numeri 37
Addizione di numeri 38
Addizione di due byte 41
Sottrazione di numeri 42
Sommario 44

CAPITOLO 6

Controllo del programma 47

Sprite 47
Cicli con l'uso di JMP 48
Cicli infiniti 51
Confronto di numeri 53
Indirizzamento relativo 56
Sommario 57

CAPITOLO 7

Contatori, Cicli e Puntatori 59

Contatori per controllare un ciclo 59
Registri X ed Y 61
Il registro X come contatore 62
Il registro X come indice 64
Sommario 68

CAPITOLO 8

Uso di informazioni memorizzate in tabelle 71

Visualizzazione di caratteri grafici 71
Memoria grafica 72
Istruzioni sul trasferimento di registri 79
Sommario 81

CAPITOLO 9

Codici dello stato dei processi 83

Rappresentazione BCD 84
Sommario 88

CAPITOLO 10

Operatori logici e manipolatori di bit 89

Cambiamento di bit in memoria 89
Rotazione di bit in un byte 93
Moltiplicazione intelligente 96
Sommario 98

CAPITOLO 11

Dettagli sul controllo dei programmi 101

Il programma counter 101
Strutture di controllo dello Stack 103
Sottoprogrammi e Stack 104
Sommario 108

CAPITOLO 12

Lavorare con il sistema operativo 109

Il Kernal 109
CLI 110
RTI 110
Sommario 113

APPENDICI

Appendice 1 Codici delle istruzioni del 6510 117
Appendice 2 Registri del processore 6510 127
Appendice 3 Tavola di conversione esadecimale/decimale 129
Appendice 4 Tabella dei salti relativi e di numerazione in complemento a due 131
Appendice 5 Mappa di memoria del Commodore 64 133
Appendice 6 Il chip dello schermo 135
Appendice 7 Il chip del suono 145
Appendice 8 Il chip interfaccia CIA 153
Appendice 9 Uso delle locazioni di memoria 159
Appendice 10 Routine del sistema operativo 173
Appendice 11 Tavola dei codici dello schermo 189
Appendice 12 Tasto corrente premuto 191
Appendice 13 ALPA 193
Appendice 14 Tabella dei valori delle note musicali ... 207
Glossario 211

PREFAZIONE

Se avete un C64 da un po' di tempo e usate il BASIC per scrivere programmi che compiono piccoli lavori, avrete lentamente esplorato e cominciato a fare degli esperimenti con il vostro computer.
E' possibile che alcuni di voi abbiano utilizzato il computer per eseguire programmi professionali come: word processing, sistemi contabili, software educativo e giochi; tutti, comunque, vi sarete chiesti che cosa rende questi programmi cosi' diversi da quelli scritti in BASIC. I suddetti programmi sono capaci di eseguire molti lavori contemporaneamente, incluso funzioni che non avreste mai pensato che il vostro computer potesse espletare.
A prescindere dalla lunghezza e dalla quantita' di tempo perso per scriverli, la maggior differenza che esiste fra i vostri programmi e quelli che comunemente si trovano in commercio e' che questi ultimi sono scritti, interamente o in parte, in linguaggio macchina. Il linguaggio macchina e' d'obbligo per un programmatore serio; quasi tutti i giochi, le utility piu' interessanti ed i programmi di interfacciamento sono scritti in linguaggio macchina.
Questo libro prova a inserirvi nel mondo del linguaggio macchina, l'altra parte del Commodore 64. Sarete condotti all'interno del set di istruzioni del linguaggio macchina lentamente, mettendo in pratica ogni cosa imparata per mezzo del programma monitor/assembler ALPA (di cui viene fornito il listato all'interno del libro).
Muovendovi all'interno del set delle istruzioni imparerete a conoscere aspetti del computer che non sapevate possedesse. Sarete incoraggiati a controllare che il risultato prodotto dal computer sia quello che realmente aspettavate, prenderete carta e penna e segnerete il percorso svolto dal microprocessore per confrontarlo con quello che vi sareste aspettati secondo la logica del vostro programma.
Le appendici con le spiegazioni sono collocate alla fine del volumetto e spesso dovrete ricorrervi a causa dei riferimenti presenti nel libro. Il resto serve per darvi delle informazioni che vi torneranno utili anche dopo che avrete finito di leggere e usare questa pubblicazione. E' presente anche una lista dei termini usati piu' comunemente per un piu' veloce e facile accesso in caso di necessita'.

L'autore ringrazia sentitamente Paul Rosham e Carolyn Sparke per l'assistenza ricevuta nella stesura del libro.

CAPITOLO 1

INTRODUZIONE AL LINGUAGGIO MACCHINA

Uno dei vantaggi del linguaggio macchina e' quello di consentire al programmatore di svolgere funzioni per cui il BASIC non e' adatto. Il vantaggio piu' rimarchevole del linguaggio macchina e' comunque la velocita'; sul C64 si possono effettuare approssimativamente 100.000 istruzioni in L.M. al secondo, mentre i comandi BASIC sono centinaia di volte piu' lenti.
Cio' e' dovuto al fatto che il BASIC e' tradotto in linguaggio macchina e che ad ogni comando BASIC possono corrispondere centinaia di istruzioni in linguaggio macchina: questo si riflette negativamente nelle capacita' di ciascuno dei linguaggi.
Le istruzioni in linguaggio macchina, come potete vedere proseguendo nella lettura di questo libro, hanno possibilita' limitate: permettono di eseguire piccoli lavori (sono necessari molti di questi per ottenere delle funzioni utili); svolgendo lavori riguardanti lo stato attuale del computer, segnalano al computer stesso di ricordare alcuni numeri e dimenticarne altri, di vedere se un tasto della tastiera e' stato premuto, di leggere o scrivere un dato sul nastro, di stampare un carattere sullo schermo....
I programmi in linguaggio macchina possono essere pensati come dei sottoprogrammi in BASIC: un programma all'interno di un altro che quest'ultimo puo' chiamare in qualsiasi momento per poi tornare al punto di partenza quando ha finito la sua funzione. Per accedere ad un sottoprogramma e per ritornare si usano rispettivamente i comandi GOSUB e RETURN.

```
.
.
.
.
10 GOSUB 1000
.
.
.
.
10000 RETURN
.
.
```

Questo non e' un sottoprogramma molto utile perche' non fa nulla, mostra solo come opera un sottoprogramma.

Uso di un programma in linguaggio macchina.

Per chiamare un sottoprogramma in linguaggio macchina dal BASIC si usa il comando "SYS indirizzo". Alla maniera del comando GOSUB bisogna segnalare al calcolatore dove inizia il sottoprogramma. "GOSUB 1000" chiama il sottoprogramma alla linea 1000; allo stesso modo "SYS 1000" chiama il sottoprogramma in linguaggio macchina iniziante alla locazione di memoria 1000.
NOTA: la locazione di memoria 1000 e' ben diversa dalla linea 1000, essa non e' una linea di programma bensi' l'"indirizzo" di un pezzo di memoria del computer.

Indirizzamento della memoria

Voi sapete che il C64 ha 64K di memoria. 64K rappresenta il numero di pezzi individuali di memoria nel computer. Ciascun pezzo puo' essere considerato come una casella che contiene un carattere o un pezzo di un'informazione.
Con piu' di 65000 caselle il computer deve avere un sistema di archiviazione che gli permetta di trovare qualsiasi pezzo di informazione desideri. Questo sistema da' a ciascuna casella un "indirizzo", come l'indirizzo di casa vostra. Voi usate l'indirizzo per trovare una determinata casa, ricevere o mandare la posta. Il computer, come voi, manda informazioni e le muove da un posto (sottoprogramma) all'altro usando il suo sistema di indirizzamento.
Questo sistema e' piu' semplice del vostro e inizia da uno dei due capi della memoria chiamandolo indirizzo zero, poi conta le caselle in memoria, dando a ciascuna di esse un numero, da 0 per la prima a 65535 per l'ultima, che coincide con l'altro capo della memoria. Per noi tutto questo sarebbe difficile da ricordare, ma per il computer e' il modo di lavorare piu' logico. Queste caselle possono essere considerate come quelle di un ufficio postale; se si mette qualcosa all'indirizzo uno, questo rimane li' finche' non si mette qualcos'altro al suo posto.
Ogni casella puo' contenere una sola cosa per volta, se si mette qualcos'altro in una casella, quello che c'era originariamente viene perduto per sempre. Il comando "SYS 1000" dice al BASIC di eseguire un sottoprogramma la cui prima istruzione e' memorizzata nella casella di indirizzo 1000.

Uso della memoria direttamente dal BASIC

Ci sono altri 2 comandi, precisamente PEEK e POKE che

troverete estremamente utili per svolgere questo lavoro. Essi rendono possibile mettere e prendere cose direttamente nelle caselle in memoria. PRINT PEEK(500) prende il contenuto della casella alla locazione di memoria 500 e lo stampa. Si puo' usare come qualsiasi altra funzione all'interno di un programma BASIC, ad esempio:

LET A=PEEK(387) o LET C=7*PEEK(1078)+14.

POKE 1100,27 mette il numero dopo la virgola, in questo caso 27, nella casella alla locazione di memoria 1100; esempio: POKE 2179,B o POKE C,X. Provate questo:

```
PRINT PEEK(1000)
POKE (1000),200
PRINT PEEK(1000)
```

Adopereremo molto spesso questi comandi BASIC durante i nostri esperimenti con le istruzioni in linguaggio macchina, per vedere i risultati dei programmi che abbiamo scritto e usato. Il BASIC ci servira' come supporto per scrivere, far girare ed osservare i nostri programmi in linguaggio macchina.

Il linguaggio macchina come un sottoprogramma

Abbiamo detto che i nostri programmi in linguaggio macchina possono essere usati come sottoprogrammi in BASIC: basta usare "SYS" in luogo del comando "GOSUB". In BASIC, come sapete, un sottoprogramma deve terminare con il comando RETURN.

```
      .
      .
      .
      .
GOSUB 1000

      .
      .
      .
      .
1000 ...
     ...
1020 RETURN
```

Allo stesso modo anche la nostra routine in linguaggio macchina deve terminare con un comando per tornare al programma principale, ma non sara' un comando BASIC, bensi' un'istruzione in linguaggio macchina.

L'istruzione in linguaggio macchina per RETURN e' --96--. 96 e' quello che il microprocessore interpreta come un

comando RETURN da un sottoprogramma. Naturalmente sarebbe impossibile per noi ricordare perfettamente che 96 e' RETURN insieme ad altre centinaia di istruzioni; per questo motivo abbiamo dato dei nomi a ciascuna istruzione. Questi nomi non hanno alcun significato per il calcolatore, ma ne hanno moltissimo per il programmatore; sono nomi corti, semplici, facilmente comprensibili e sono chiamati Mnemonici.

Il mnemonico di 96 e' RTS e sta per RETURN da un sottoprogramma. Dove sara' necessario indicheremo il numero in codice macchina ed il mnemonico di un'istruzione, per renderla piu' leggibile e allo stesso tempo fornire al computer le informazioni che gli occorrono.

Per dimostrare quello che e' stato detto abbiamo fatto un piccolissimo programma in linguaggio macchina. Scrivete la seguente linea BASIC:

```
POKE 49152,96
```

Questa mette 96 (il valore numerico dell'istruzione RTS) nella casella alla locazione di memoria 49152.

Congratulazioni, avete appena creato il vostro primo programma in linguaggio macchina! Non e' molto, e' come il sottoprogramma BASIC:

```
GOSUB 400
400 RETURN
```

Poiche' adesso l'istruzione 96 (RTS) e' nella casella alla locazione di memoria 49152, facciamo girare il programma per vedere come lavora usando il comando "SYS".
Scrivete la seguente linea BASIC:

```
SYS 49152
```

Il computer rispondera' con READY. Ha appena eseguito il vostro programma.

Capitolo 1 SOMMARIO

1. Il codice macchina (assembler) e' piu' veloce e permette di accedere a funzioni insite nell'hardware del calcolatore che non e' conveniente usare dal BASIC.
2. I comandi possono eseguire molte meno funzioni.
3. La memoria e' indirizzabile usando numeri da 0 a 65535.
4. Una locazione di memoria puo' essere paragonata ad una casella postale che puo' contenere un solo pezzo di informazione per volta.
5. Il comando PEEK viene usato dal BASIC, per esaminare il contenuto di una locazione di memoria.

6. Il comando POKE viene usato dal BASIC per mettere qualcosa in una locazione di memoria.
7. Il comando SYS viene usato dal BASIC per eseguire un programma in linguaggio macchina.
8. Il valore 96 (RTS) deve essere messo alla fine di ogni programma in linguaggio macchina per segnalare al computer "RETURN (tornare) dal sottoprogramma".

CAPITOLO 2

BASI DELLA PROGRAMMAZIONE IN LINGUAGGIO MACCHINA

Come il linguaggio macchina usa la memoria

Abbiamo discusso a lungo della memoria, come vi si possano cercare o come vi si possano memorizzare cose dal BASIC.

Naturalmente cio' si puo' fare ugualmente bene con il linguaggio macchina; dobbiamo cioe' essere capaci di prelevare delle informazioni da una delle caselle di memoria, svolgere delle operazioni sul suo contenuto e rimemorizzarle nella stessa o in differenti caselle. Per tutte queste operazioni il microprocessore possiede degli strumenti chiamati registri, che possiamo paragonare a delle mani con cui il microprocessore compie le cose suddette.

I registri

Ci sono 3 tipi di registri (mani) ed ognuno e' adatto per svolgere un compito differente, alla stessa maniera con cui noi usiamo le mani (per esempio per giocare a tennis, dove la destra solitamente tiene la racchetta e la sinistra la palla, oppure per legarci le scarpe dove le usiamo entrambe).

Queste mani (registri) prendono le informazioni dalla memoria e come essa non possono contenere piu' di un'informazione per volta; ma non sono esse stesse una parte della memoria ne' hanno alcun indirizzo. Sono una parte speciale del microprocessore e sono necessarie istruzioni specifiche che operano con loro separatamente.

L'accumulatore

Il primo registro di cui tratteremo e' il registro 'A' (o Accumulatore) che, come potrete vedere nei capitoli seguenti, e' la mano che compie le funzioni piu' generali del computer ed anche il registro che sviluppa la maggior parte delle operazioni matematiche.

Il piu' delle volte il microprocessore deve avere delle informazioni nei registri prima di poter operare con essi; per far prendere informazioni dalla memoria, utilizzando

l'accumulatore, si usa l'istruzione "LDA". Questo codice mnemonico sta' per LOAD (carica in) accumulatore; infatti carica il contenuto di una delle caselle presenti in memoria nell'accumulatore del microprocessore. Esempio: LDA 253.
Questo comando carica in accumulatore il contenuto della cella di memoria all'indirizzo 253. Il codice macchina di questo comando e' 165 253.
NOTA: il codice macchina e' in due parti; a differenza del comando RTS, -96- il comando LDA 253 e' in due parti; la prima per LDA e' uguale a -165-, la seconda parte riguarda l'indirizzo della cella di memoria che contiene l'informazione ed e' -253-. Queste parti separate della stessa istruzione vengono messe in celle di memoria distinte in modo che le caselle

contenenti il programma:	LDA 38 RTS
vengono viste come:	165
	38
	96

Metodi di indirizzamento

Molte istruzioni in linguaggio macchina hanno differenti forme e metodi che consentono al programmatore una certa flessibilita' nella scelta di come e dove mettere i dati per far lavorare il suo programma. Soltanto per l'istruzione LDA ci sono 8 differenti forme dette Metodi di indirizzamento.

I metodi di indirizzamento cambiano e specificano, all'interno dell'istruzione, il modo in cui deve essere usato l'indirizzo della cella di memoria.
Supponiamo, per esempio, di avere un'istruzione che dice di prendere una lettera da una certa casella postale; cio' puo' essere fatto in modi diversi:

1 Potrebbe dire di cercare la casella 17.
2 Potrebbe dire di cercare la terza casella da destra sulla seconda riga.
3 Potrebbe dire di cercare la casella di proprieta' del Sig. Rossi.
4 Potrebbe dire di cercare la casella il cui indirizzo e' contenuto in un'altra.
5 Potrebbe dire semplicemente di prendere la lettera.

Tutto quello che riguarda i metodi di indirizzamento lo

vedremo piu' avanti nel libro, per adesso introduciamo solo 3 delle 8 differenti forme del comando LDA.

Metodo 1 - 165 253 LDA 253

Questa e' una forma corta di LDA. Per motivi che spiegheremo piu' avanti essa puo' accedere ad un piccolo range di indirizzi di memoria e viene chiamata Indirizzamento in pagina zero.

Metodo 2 - 173 55 4 LDA 1079

E' la forma piu' lunga del comando LDA e puo' accedere ad una casella in qualunque punto della memoria. NOTA: qui il codice macchina e' diviso in tre parti. La prima, -173-, e' il comando LDA in questa forma; la seconda, -55- e la terza -4-, rappresentano l'indirizzo della casella 1079 che contiene il dato da caricare nel registro A. La ragione per cui il numero 1079 viene suddiviso in 55 e 4 vi sara' piu' chiara procedendo nella lettura dei prossimi capitoli. Questo metodo viene chiamato Indirizzamento assoluto.

Metodo 3 - 169 71 LDA #71

Il comando e' diverso dai due precedenti; invece di cercare l'informazione da caricare nell'accumulatore in una delle celle di memoria, l'informazione desiderata viene data come parte dell'istruzione. In questo caso il numero 71 viene caricato nell'accumulatore ed equivale all'esempio numero 5 proposto per la casella postale. Fate attenzione: questo tipo di indirizzamento, detto Indirizzamento immediato, viene visto in codice mnemonico con il numero sempre preceduto dal simbolo (#) cancelletto.

Adesso conosciamo come il microprocessore preleva qualcosa dalla memoria ma, per poter fare qualsiasi operazione, e' necessario conoscere come fargli fare qualcosa con quello che ha prelevato. Per far mettere in memoria quello che ha nella mano A (accumulatore), il microprocessore usa l'istruzione STA, che sta per Store (memorizza) l'accumulatore. Questa istruzione mette il contenuto dell'accumulatore in una specifica cella di memoria. Anche questa istruzione ha piu' modi di indirizzamento (7), ma noi ne vedremo in questo momento solo 2.

Metodo 1 - 133 41 STA 41

Questa istruzione mette il contenuto dell'accumulatore nella cella di indirizzo 41. Essa, come l'analoga istruzione del comando LDA, e' divisa in due parti (metodo in pagina zero) e puo' accedere ad un numero limitato di indirizzi di celle in memoria.

Metodo 2 - 141 57 03 STA 825

E' identico al metodo 1, con la sola differenza che puo' mettere il contenuto dell'accumulatore in qualsiasi parte della memoria (indirizzamento assoluto). Il numero -141- identifica l'istruzione, i numeri -57- e -3- contengono l'indirizzo della cella 825 (vedi capitolo 3 per maggiori chiarimenti).
DOMANDA: Perche' non esiste per l'istruzione STA un metodo immediato (vedi LDA#71)?
RISPOSTA: Il metodo immediato in "LDA#71" mette il numero - 71 - contenuto nell'istruzione, direttamente nell'accumulatore come se prendessimo una lettera, non gia' un numero di casella postale in cui prelevare una lettera. Il metodo immediato per STA significherebbe mettere il contenuto dell'accumulatore nell'istruzione stessa: sarebbe come se mettessimo una lettera non in una casella postale, ma nelle istruzioni che ci hanno dato per essa e questo, ovviamente, non avrebbe senso.

Una semplice routine che carica dei dati in memoria.

Adesso scriveremo un piccolo programma in linguaggio macchina per esaminare le istruzioni appena imparate; per rendere tutto piu' facile battete il seguente programma BASIC:

```
5    PRINT CHR$(147);"...."
10   REM QUESTO PROGRAMMA RENDE SEMPLICE
     L'INSERIMENTO DI PROGRAMMI IN
     CODICE MACCHINA
20   READ A
30   IF A=-1 THEN GOTO 70
40   POKE 49152+X,A
50   X=X+1
60   GOTO 20
```

```
70   PRINT"PRIMA...-LOCAZIONE 1024 ";PEEK(1024)
80   SYS 49152
90   PRINT"DOPO....-LOCAZIONE 1024 ";PEEK(1024)
100  END
1000 DATA 169,1: REM LDA #1
1010 DATA 141,0,4: REM STA 1024
1020 DATA 96: REM RTS
9999 DATA -1
```

Linee 1000 - 9999 contengono il programma in linguaggio macchina.
Linee 20 - 60 mettono il programma in memoria a partire dalla cella 49152.
Linee 70 - 90 fanno il test sulla cella di memoria che abbiamo deciso di cambiare.

Quando il programma BASIC ha finito di lavorare, il nostro programma in linguaggio macchina e' contenuto in memoria come segue:

Indirizzo	Dato
49152	169
49153	1
49154	141
49155	0
49156	4
49157	96

In forma mnemonica:

```
49152    LDA#1
49154    STA 1024
49157    RTS
```

Linguaggio simbolico

Un programma scritto in forma mnemonica e' chiamato programma in linguaggio "assembler" (simbolico); ma per tradurre la lista di lettere comprensibili al programmatore in una lista di lettere comprensibili al microprocessore si usa un programma chiamato "Assembler" (assemblatore).

Proseguendo nella lettura del libro troverete i programmi in entrambi i formati:

INDIRIZZO	CODICE			MNEMONICO
49152:	169	1		LDA#1
49154:	141	Ø	4	STA 1Ø24
49157:	96			RTS

Il nostro programma BASIC, oltre a porre il codice macchina in memoria, lo fa eseguire (vedi linea 8Ø).

Come potete vedere dai due messaggi PRIMA e DOPO che analizzano l'indirizzo di memoria 1Ø24, quest'ultimo e' stato cambiato dal nostro programma. Il valore originale della locazione 1024 non ha alcuna importanza e puo' cambiare ogni volta che si esegue il programma. E' impossibile conoscere cosa ci sara' in memoria prima che ci abbiate messo qualcosa, proprio come voi non potete dire cosa c'e' in una casella postale prima di averci guardato dentro. Il valore della cella 1024 dopo aver eseguito il programma e' 1; questo dimostra che il nostro programma ha fatto quello che ci aspettavamo: ha caricato in accumulatore il numero 1 e poi lo ha memorizzato alla cella 1024.

Memoria di schermo

Questo e' uno dei risultati che non vi aspettavate dal nostro programma; guardate l'angolo sinistro in alto dello schermo e vedrete stampata una "A". La linea 5 del programma BASIC puliva lo schermo e in nessun altro posto del programma c'era un'istruzione che stampasse una "A"; quindi deve essere stata messa li' dal programma in codice macchina. Noi sappiamo che il programma in linguaggio macchina mette il valore 1 nella cella 1024. Vuol forse dire di stampare una "A" sullo schermo? Provate dal BASIC e vedete cosa succede. Premete il tasto CLR per cancellare lo schermo e fate:

```
POKE 1024,1
```

Vedrete di nuovo una "A" apparire sullo schermo nell'angolo sinistro in alto; cio' succede perche' la memoria a 1024 compie due funzioni: visualizza le cose sullo schermo e svolge una funzione di riconoscimento per la memoria normale. La similitudine dell'ufficio postale e'

ancora valida ma, adesso, la casella sembra avere una vetrata nella parte posteriore attraverso la quale potete vedere cosa contiene. Se sfogliate la tebella dei codici dello schermo nell'appendice 12, noterete che per il valore 1 che noi abbiamo messo alla locazione 1024 viene stampata la lettera "A". (Il SET1, maiuscolo, viene usato per default. Per passare al set minuscolo bisogna premere i tasti SHIFT e "Commodore" contemporaneamente).

Proviamo a stampare altri caratteri contenuti in tabella, ad esempio una "X". Per prima cosa dobbiamo cercare in tabella il valore del codice dello schermo associato alla lettera "X", che e' 24. Per metterlo in memoria usiamo di nuovo il nostro programma:

```
LDA #1
STA 1024
RTS
```

ma questa volta cambiamo LDA #1 con LDA #24. Usando lo stesso programma BASIC di prima, carichiamolo in memoria, avendo pero' l'avvertenza di cambiare prima la linea 1000, contenente i dati per il comando LDA, che deve essere letta:

```
1000 data 169,24      :REM LDA #24
```

Il nostro programma in linguaggio macchina, dopo che e' stato fatto eseguire il programma BASIC, diventa cosi':

```
49152   169   24       LDA #24
49154   141   0    4   STA 1024
49157   96    0        RTS
```

Eseguito questo programma apparira' una "X" nell'angolo sinistro in alto dello schermo.

A questo punto potreste chiedere: "Come si fa' a stampare qualcosa in qualsiasi punto dello schermo"? La risposta e' semplice: la "memoria di schermo" esiste in memoria dall'indirizzo 1024 all'indirizzo 2023 e consta di 25 righe e 40 colonne come si puo' facilmente vedere sullo schermo. La memoria a 1024 appare nell'angolo sinistro in alto, 1025 e' di seguito sulla destra, 1026 a destra ancora e cosi' via. In questo modo 1024 + 40 (1064) corrisponde alla posizione immediatamente sotto 1024, precisamente sul margine sinistro della seconda riga; 1064 + 40 (1104) corrisponde alla posizione ancora sotto e cosi' via.

Usando il nostro solito programma BASIC cercheremo di stampare sulla seconda riga a partire dall'alto. L'indirizzo di questa posizione e' dato da 1024 + 40 (base schermo + 1 riga)= 1064.
Percio' il nostro programma diventa:

```
LDA #24     Lettera "X"
STA 1024    Prima colonna seconda riga
RTS
```

Per far cio' cambiamo i dati per il nostro programma alla linea 1010:

```
1010 DATA 141,40,4      :REM STA 1064
```

Il programma in linguaggio macchina stampera' in questo modo una "X" sulla seconda riga dello schermo a partire dall'alto.

Stampa di un messaggio

Adesso possiamo usare il nostro programma BASIC per scrivere un programma piu' ampio che scrivera' un messaggio sullo schermo. Scrivete le seguenti linee:

```
1000  DATA  169,8
1010  DATA  141,0,4
1020  DATA  169,5
1030  DATA  141,1,4
1040  DATA  169,12
1050  DATA  141,2,4
1060  DATA  141,3,4
1070  DATA  169,15
1080  DATA  141,4,4
1090  DATA  96
```

Fate eseguire il programma e vedrete stampata la scritta "HELLO" in alto sullo schermo. Il programma in codice macchina che abbiamo usato per stampare quel messaggio era:

Indirizzo	Codice macchina			Codice assembler	
49152	169	8		LDA # 8	Codice di schermo per "H"
49154	141	Ø	4	STA 1Ø24	
49157	169	5		LDA # 5	Codice di schermo per "E"
49159	141	1	4	STA 1Ø25	
49162	169	12		LDA # 12	Codice di schermo per "L"
49164	141	2	4	STA 1Ø26	
49167	141	3	4	STA 1Ø27	
49171Ø	169	15		LDA # 15	Codice di schermo per "O"
49172	141	4	4	STA 1Ø28	
49175	96			RTS	

Controllate i valori usati con quelli nell'apposita tabella. E' interessante notare il modo con cui vengono stampate le due "L". Non e' stato necessario ricaricare il valore 12 nell'accumulatore dopo che questo valore era stato messo in memoria. Cio' succede perche', ogniqualvolta prendiamo qualcosa dalla memoria, oppure quando vi mettiamo il contenuto di uno dei registri, viene fatta una copia dell'originale mentre quest'ultimo rimane dov'e'.

Possiamo scrivere il programma che abbiamo appena finito usando differenti metodi di indirizzamento; cio' risulta molto utile per ragioni di efficienza del programma stesso. Puo' capitare che si abbia bisogno di un programma il piu' veloce possibile, oppure il piu' corto, o ancora che sia comprensibile e si corregga facilmente.

Cambieremo allora il nostro programma, in modo da avere una grande flessibilita' nelle cose che stampa. Scrivete le seguenti linee:

```
15   INPUT"VALORE LETTERA ";B:POKE 252,B
1000 DATA 165,252            :REM LDA 252
1090 DATA 169,23             :REM LDA #23
1100 DATA 141,5,4            :REM STA 1029
1110 DATA 96                 :REM RTS
```

Il nostro programma in linguaggio macchina diventa:

Indirizzo	Codice macchina			Codice assembler
49152	165	252		LDA 252
49154	141	Ø	4	STA 1Ø24
49157	169	5		LDA # 5
49159	141	1	4	STA 1Ø25
49162	169	12		LDA # 12
49164	141	2	4	STA 1Ø26
49167	141	3	4	STA 1Ø26
4917Ø	169	15		LDA # 15
49172	141	4	4	STA 1Ø28
49175	169	23		LDA # 23
49177	141	5	4	STA 1Ø29
4918Ø	96			RTS

NOTA: Il programma trova il valore della sua prima lettera alla cella di memoria 252 usando l'indirizzamento in pagina zero invece di quello immediato. La linea 15 del programma BASIC fa si' che questa cella contenga qualsiasi numero si voglia. Eseguite questo programma piu' volte usando i valori 25,2 e 13.

In questo capitolo abbiamo visto come la memoria puo' avere piu' di una funzione (ad esempio la memoria fra 1024 e 2023); nello stesso modo altre parti di memoria svolgono piu' funzioni. Aree di memoria sono usate per controllare i colori dello schermo, la grafica, gli sprite, il suono,la tastiera, i joistick e molte altre funzioni di I/O. Di queste aree tratteremo a un livello puramente introduttivo; per saperne qualcosa di piu' sarebbe meglio ricorrere a testi piu' avanzati, che trattano questi argomenti piu' profondamente, come ad esempio: "Commodore 64 Exposed" di Bruce Bayley pubblicato dalla Melbourne House, nonche' al manuale fornito all'atto dell'acquisto della macchina.

Capitolo 2 SOMMARIO

1. Il microprocessore usa i registri per muovere le cose nella memoria e lavorarci sopra.
2. Ci sono tre registri multiuso (A "accumulatore", X e Y).
3. Il comando LDA si usa per caricare qualcosa nell'accumulatore.
4. Il comando STA serve per mettere il contenuto dell'accumulatore in memoria.

5. Questi comandi insieme a molti altri hanno diversi metodi di indirizzamento che ci danno una certa flessibilita' nel memorizzare e usare i dati:

- indirizzamento immediato - carica il dato contenuto nell'istruzione,
- indirizzamento assoluto - usa dati memorizzati in qualsiasi punto della memoria,
- indirizzamento in pagina zero - utilizza i dati contenuti in un'area limitata di memoria.

6. Un programma scritto in forma mnemonica viene chiamato programma in codice assembler.
7. La memoria viene usata per stampare informazioni sullo schermo.
8. Le informazioni sono stampate in base a dei codici di schermo che danno un valore numerico a ogni carattere stampabile.
9. La memoria viene usata per controllare altre funzioni di I/O (input/output) del calcolatore.

CAPITOLO 3

INTRODUZIONE ALL'ESADECIMALE

Gli usi dell'esadecimale

Precedentemente abbiamo parlato della memoria, ma non abbiamo mai specificato che cosa puo' o non puo' contenere. Abbiamo usato la memoria per caricare numeri che rappresentano caratteri, valori numerici, istruzioni in codice macchina e indirizzi di memoria; abbiamo inserito i numeri voluti senza pensare come il computer li memorizza, accennando appena al caso dell'indirizzamento assoluto, che ci ha mostrato che il modo di numerazione del calcolatore non e' cosi' semplice come si potrebbe pensare. Per esempio 141 5 4 e' il codice macchina dell'istruzione STA 1029. Il numero 141 rappresenta l'istruzione STA e i due numeri restanti indicano l'indirizzo 1029, ed e' ovvio che c'e' qualcosa su cui non ci siamo soffermati.

Abbiamo precedentemente paragonato i registri e la memoria del microprocessore a delle mani; qual e' il numero piu' grande che si puo' rappresentare con una mano? Bene, cio' dipende da cosa intendiamo per rappresentare. Con le dita di una mano si puo' contare fino a cinque, cioe' rappresentare un numero da 0 a 5. Questo significa che il piu' grande numero che possiamo rappresentare e' 5? Sarete sorpresi di sapere che la risposta e' no.

Contare da 0 a 5 in questo modo:

0 1 2 3 4 5

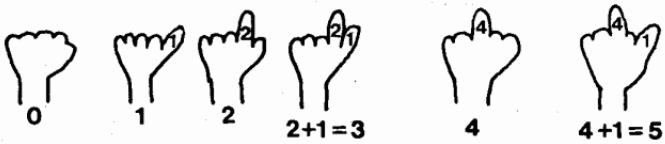
e' un modo che limita le risorse della mano, come lo sarebbe per il computer.

Binario

Le "dita" di un calcolatore possono essere alzate o abbassate (accese o spente) ma, come voi per le mani, cosi' esso puo' dire quale delle "dita" e' accesa o spenta. In altre parole, il numero rappresentato dipende non dal

numero di dita, ma dalla posizione di queste ultime. Provate da soli: date a ciascun dito i seguenti valori (se volete, scriverli sulle dita con una penna).

Adesso contate sommando i numeri rappresentati da ciascun dito nella sua posizione:



Cercate di rappresentare questi numeri sulle dita: 7, 16, 10, 21, 29.

D. Qual e' il numero piu' grande che potete rappresentare sulle dita?
R. 1+2+4+8+16=31

Come vedete, dalla rappresentazione originale di 5, siamo gia' arrivati a rappresentare il numero 31 (di piu' non si puo' fare con una sola mano). Le "mani" del calcolatore pero' differiscono in tante cose dalle nostre; sono innanzitutto dei segnali elettrici e possono essere sia accesi che spenti. A beneficio del programmatore diciamo che la condizione di spento e' data dal valore 0 e la condizione di acceso e' data dal valore 1. L'altra piu' vistosa differenza e' che il computer ha 8 "dita" per ogni mano; cio' puo' sembrare strano, ma non c'e' nessuna ragione per cui non debba essere cosi'. La "mano" a 8 "dita" del computer viene chiamata un "byte" di memoria. Come abbiamo fatto prima per le nostre dita, assegnamo a quelle del calcolatore i valori: 1, 2, 4, 8, 16, 32, 64, 128.

Ricontiamo di nuovo i valori di tutte le dita nella posizione attivata (accesa).

Mano a otto dita	"Mano" del calcolatore -byte	Numero
	0 0 1 1 0 0 0 1	32+16+1 = 49
	1 1 0 0 0 1 0 0	128+64+4 = 196
	0 0 0 1 0 0 0 1	16+1 = 17

D. Qual e' il massimo numero rappresentabile dalle 8 "dita" del computer?

R. 128+64+32+16+8+4+2+1 = 255.

Senza rendercene conto abbiamo introdotto in questo capitolo il sistema di numerazione binario (base 2). Tutti i calcolatori operano in base 2, rappresentando le fasi elettriche accesa e spenta in una serie infinita di 1 e di 0. Cio' renderebbe piu' difficile al programmatore capire quello che succede all'interno di un calcolatore. Esempio:

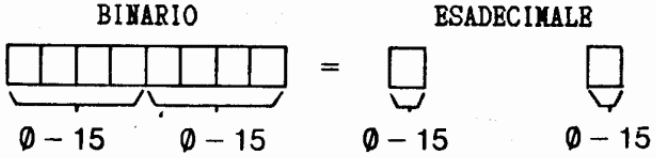
Codice assembler	Codice macchina	Binario	
LDA #8	169 8	10101001	00001000
STA 1029	149 4	10010101	00000100
	5	00000101	
RTS	96	01100000	

Perche' esadecimale?

Tutto cio' pero' sarebbe impossibile da scrivere e da ricordare correttamente al momento della stesura del programma da parte del programmatore; potremmo usare ancora la forma decimale, ma non sarebbe quella piu' opportuna per capire il modo in cui lavora il calcolatore. Percio' viene usata la forma esadecimale o in base sedici. Al primo impatto puo' sembrare difficile da usare, ma in seguito si rivela la forma piu' adatta e quella piu' vicina alla rappresentazione binaria che viene usata dal computer per memorizzare i dati.

La conversione fra binario ed esadecimale e' molto semplice. Ogni cifra esadecimale rappresenta un numero fra

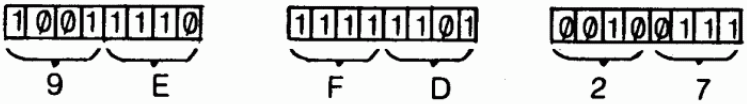
0 e 16, allo stesso modo in cui una cifra decimale rappresenta un numero compreso fra 0 e 9; inoltre ogni cifra decimale rappresenta la meta' esatta di un byte (mano a sedici dita).



e l'intero byte e' rappresentato, quindi, da due cifre esadecimali. Adesso vi chiederete come una sola cifra possa rappresentare un numero fra 0 e 15: esso, e' identico al decimale e differisce nel fatto che i numeri 10, 11, 12, 13, 14 e 15 sono rappresentati rispettivamente dalle lettere A, B, C, D, E e F.

BINARIO	DECIMALE	ESADECIMALE
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
10000	16	10

La prossima figura mostra che la conversione da binario a esadecimale e' facilmente ottenibile dividendo il byte in due segmenti da 4 bit (dita) ciascuno.



Matematica esadecimale e binaria

In matematica ogni base segue delle semplici regole di formato; ogni cifra assume il valore: CIFRA*(BASE)POSIZIONE-1

In altre parole 98617 in decimale e':

$$7 \times 10^0 + 1 \times 10^1 + 6 \times 10^2 + 8 \times 10^3 + 9 \times 10^4 = 98617$$
$$7 \times 1 + 1 \times 10 + 6 \times 100 + 8 \times 1000 + 9 \times 10000 = 98617$$
$$7 + 10 + 600 + 8000 + 90000 = 98617$$

01011101 in binario e':

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 0 \times 2^7 = 93$$
$$1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8 + 1 \times 16 + 0 \times 32 + 1 \times 64 + 0 \times 128 = 93$$
$$1 + 0 + 4 + 8 + 16 + 0 + 64 + 0 = 93$$

A7C4E in esadecimale e':

$$14 \times 16^0 + 4 \times 16^1 + 12 \times 16^2 + 7 \times 16^3 + 10 \times 16^4 = 687182$$
$$14 \times 1 + 4 \times 16 + 12 \times 256 + 7 \times 4096 + 10 \times 65536 = 687182$$
$$14 + 64 + 3072 + 28672 + 655360 = 687182$$

Vi sono alcuni punti, in quello detto finora, su cui dovremmo soffermarci. Per prima cosa, ogni numero che puo' essere memorizzato in una cella di memoria (un numero compreso fra 0 e 255) e' rappresentabile in otto cifre binarie (bit) o, come abbiamo detto precedentemente, "dita". Ogni numero compreso fra 0 e 255 puo' pero' anche essere rappresentato da due cifre esadecimali (FF = 15 X 16↑1 + 15 X 1 = 255). A questo punto nasce il problema dell'indirizzamento assoluto di cui avevamo accennato: se il massimo numero rappresentabile in un byte di memoria e' minore di 256, come faremo a specificare un indirizzo compreso fra 0 e 65535 (64K)? La soluzione e' quella di usare due celle di memoria, non sommandole insieme, ma intese come parti dello stesso numero; questo implica che gli indirizzamenti siano effettuati con numeri binari a 16 bit ("dita")(2 byte), che e' come dire numeri esadecimali a 4 cifre. Il numero massimo rappresentabile con 4 cifre esadecimali e':

```
FFFF=15X1+15X16+15X256+15X4096
    =15+240+3840+61440
    =65535=64K
```

che e' abbastanza grande per indirizzare tutta la memoria.

Esempio: il numero esadecimale da 2 byte 13A9 equivale a:

1	3	A	9
0001	0011	1010	1001

$$13 \times 16^2 + A9 \times 16^0$$
$$13 \times 256 + A9$$
$$= 4864$$

Altro esempio il numero esadecimale da 2 byte 0405

=4X256+5
=1024+5
=1029

Indirizzamento assoluto

Come abbiamo detto precedentemente esisteva un problema legato a questo tipo di indirizzamento e lo abbiamo risolto, ma c'e' un'altra cosa molto importante da ricordare che riguarda l'ordine con cui sono memorizzati i byte che compongono l'indirizzo: essi sono sempre memorizzati in ordine inverso e cioe':

LDA 1029
= 141 5 4

Il byte piu' significativo (byte alto) 4 e' messo dopo il byte meno significativo (byte basso) 5. NOTA: e' l'opposto della forma normale; esempio: prendiamo il numero 17, il byte piu' significativo (1X10) e' memorizzato prima e quello meno significativo (7X1) dopo. Le ragioni per cui, con gli indirizzamenti assoluti, si usa la configurazione byte basso - byte alto non staremo qui a specificarle.

In questo capitolo parleremo anche di un altro tipo di indirizzamento, quello cosiddetto in "pagina zero". L'istruzione occupa due byte e ne lascia solo uno per specificare l'indirizzo: esempio LDA 38 -165 38. Come abbiamo visto prima, con un solo byte si possono rappresentare solo numeri inferiori a 256; quindi con l'indirizzamento in pagina zero si puo' accedere solo ai primi 256 byte di memoria. Un blocco di 256 byte viene chiamato pagina, da qui il nome dato al tipo di indirizzamento.

Per specificare che si stanno usando numeri esadecimali useremo la notazione standard che fa precedere il segno $

(dollaro) prima di ogni numero esadecimale.

```
LDA 1024  e' lo stesso di  LDA $400
LDA 65535     "    "       LDA $FFFF
LDA 0         "    "       LDA $0
```

Da adesso in poi tutti i listati in codice macchina saranno stampati in esadecimale:

Indirizzo	Codice macchina	Codice Assembler
49152	A9 8	LDA #$8
49154	8D 0 4	STA $400
49157	A9 53	LDA #$53
49159	8D 1 4	STA 1025
49162	60	RTS

in accordo con il formato usato nel codice assembler e rispettandone le varie applicazioni.

Conversione da esadecimale a decimale

Nell'appendice 3 troverete una tabella che vi aiutera' per una rapida conversione da esadecimale a decimale. Per conversioni di numeri a singolo byte guardate la colonna verticale per la prima cifra esadecimale e la riga orrizzontale per la seconda. Esempio:

```
$2A - terza riga in basso
      11-esima colonna da sinistra
C'e' stampato:  LO    HI
                42    10752
```

Guardate sotto la scritta LO (byte basso), 42 e' il corrispondente decimale di $2A esadecimale. Per numeri da due byte guardate sotto la scritta HI per il byte di sinistra (byte alto) e sommatelo al byte basso LO. Esempio:

```
$7156  scomposto  HI=$71  LO=$56
HI-71- ottava riga in basso
       seconda colonna da sinistra
LO     HI
113    28928
LO-56- sesta riga in basso
       settima colonna da sinistra
```

LO	HI
86	220₁₆

Sommate basso e alto 28928+86=29014
$7156=29014

NOTA: in tutti i casi

LO	HI
X	Y

$$Y = 256 * X$$

Il byte alto e' 256 volte il valore dello stesso byte basso.

Capitolo 3 SOMMARIO

1. Contando con le "dita" di un calcolatore, la posizione e' tanto importante quanto il numero del dito.
2. Ogni "mano" del computer, ovvero ogni pezzo di memoria, ha 8 dita e il massimo numero che puo' contenere e' 255.
3. Un pezzo di memoria a 8 "dita" e' chiamato byte.
4. Ogni "dito" ha un valore dipendente dalla sua posizione. Valore=posizione -1 1, 2, 4, 8, 16, 32, 64, 128 Binario.
5. L'esadecimale (base 16) e' come un raggruppare insieme dei binari. 1 cifra esadecimale = 4 cifre binarie. L'esadecimale e' piu' semplice da maneggiare del decimale e del binario.
6. DECIMALE 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 18
   ESADECIMALE 0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 ecc.
7. L'indirizzamento in pagina zero accede ai primi 256 byte, il massimo indirizzabile con un byte.
8. L'indirizzamento assoluto puo' accedere a 65536 (64) di memoria (tutta), che e' il massimo indirizzabile con due byte.
9. Nell'indirizzamento assoluto il byte basso e' sempre memorizzato prima di quello alto. Esempio:
   8D 98 17 LDA $1798
10. I numeri esadecimali sono specificati premettendo loro il segno $ (dollaro).
11. Ricordate la tabella per la conversione rapida da esadecimale a decimale nell'appendice 3.

CAPITOLO 4

INTRODUZIONE ALL'ALPA

Per aiutarvi a caricare in memoria i programmi in linguaggio macchina, abbiamo inserito nell'appendice 13 il listato di un assemblatore BASIC chiamato ALPA (Assembly language programming aid) (e' il programma registrato sulla cassetta allegata). Nel capitolo secondo abbiamo usato un piccolo programma BASIC per mettere il programma in L.M. (linguaggio macchina) in memoria ma, come potete immaginare, utilizzare metodi di questo tipo e' molto noioso. Da adesso in poi, tutti gli esempi di programmi in L.M. saranno scritti nel formato ALPA. I vantaggi dell'ALPA sono:

1 Il programma viene memorizzato come testo ed in una forma eseguibile con il comando ENTER.
2 Le istruzioni possono essere inserite o cancellate senza danneggiare le altre presenti in memoria.
3 Per agevolare le funzioni di inserimento, cancellazione e preparazione, ciascuna istruzione e' messa su linee separate con un numero di linea a cui riferirsi (come un normale programma BASIC).
4 Il programma inserito puo' essere listato una pagina alla volta con il comando LIST.
5 Di ogni istruzione inserita, anche in forma esadecimale, viene mostrato sullo schermo il "disassemblato" in codice assembler, per renderne piu' facile la correzione e la comprensione.
6 Con ALPA si puo' disassemblare la memoria oppure stampare il codice ASCII o l'esadecimale di qualsiasi parte di memoria.
7 Un programma puo' essere locato in qualsiasi posto in memoria.
8 E' inclusa la possibilita' di riferirsi ad istruzioni usando il numero di linea.
9 Si possono memorizzare (SAVE) o caricare (LOAD) programmi su e da nastro.

Uso dell'ALPA

Tutti i numeri usati, ad eccezione dei numeri di linea, devono essere esadecimali. Quando il programma parte, dopo essersi autoinizializzato, chiede con:

A QUALE INDIRIZZO:?

l'indirizzo di partenza del programma che volete scrivere. Come vedrete piu' avanti, molti dei nostri programmi sono stati scritti con l'intenzione di memorizzarli a partire dall'indirizzo $C000 (49152). Percio' rispondete con C000 e, se per caso fosse necessario cambiare, vi verra' detto di usare il comando CHANGE (vedi comandi di ALPA). A questo punto il calcolatore e' in attesa di un vostro inserimento e stampa il messaggio:

COMANDO O NUMERO DI LINEA (###)

Adesso si puo' inserire il programma, che deve essere scritto nel seguente formato:

linea # (SPAZIO) Byte ESA (Byte ESA)(Byte ESA)

dove - linea # e' un numero decimale compreso fra 1 e 199.
- Byte ESA e' il valore esadecimale del comando da scrivere, seguito da uno o due byte esadecimali come operandi. Esempio:

10 A905
oppure
20 8D3412

Quando scrivete una linea in questo modo, il calcolatore "disassembla" il codice; in altre parole il computer prende il byte inserito, lo stampa e ne stampa l'istruzione corrispondente in linguaggio assembler. Per esempio, se scrivete:

10 A905

il calcolatore rispondera' con:

10:A9 05 LDA #$05

Cio' permette di leggere la versione del programma in codice macchina e in formato assembler immediatamente. Percio' i programmi che troverete andando avanti nella lettura del volume saranno nel seguente formato:

```
10 A905     LDA #$05
20 8D3412   STA $1324
```

Sulla sinistra viene stampato il codice macchina e sulla destra il codice assembler corrispondente.
Il sistema di numerazione delle linee usato da ALPA e' lo stesso del BASIC:

linea ### codice
- mette il codice alla linea ### sostituendolo a quello che c'era.
linea ###
- cancella la linea ###

I comandi ALPA

1 - LIST o LIST linea ###
Lista le prime 22 linee di programma dall'inizio o a partire dalla linea ### specificata.
2 - ENTER (Non e' il tasto ENTER!)
Memorizza il programma in memoria all'indirizzo specificato alla domanda "A QUALE INDIRIZZO:?". NOTA: E' necessario fare l'ENTER di un programma prima di dargli il RUN.
3 - RUN
Esegue il programma in memoria a partire dal primo indirizzo che e' stato inserito. Il comando ENTER deve essere fatto prima di un comando RUN.
4 - WATCH
Chiede l'indirizzo che si vuole "vedere". L'indirizzo scelto viene stampato prima e dopo il RUN del programma; serve per vedere i risultati ottenuti con un programma.
5 - NWATCH
Disabilita il comando precedente.
6 - CHANGE
Ripete la domanda sull'indirizzo di partenza del programma in memoria; serve per cambiare questo indirizzo senza dover far ripartire l'ALPA.
7 - MEMORY
Pone questa domanda "DISASSEMBLO A PARTIRE DA:?" chiedendo l'indirizzo di partenza per disassemblare (produrre un codice assembler) una zona di memoria. Stampa una videata a partire dall'indirizzo di cui sopra; ogni tasto, ad esclusione di M, stampa la videata successiva. M fa tornare alla fase accetta comandi.
8 - DUMP
Come il precedente, chiede l'indirizzo di partenza da cui stampare il codice esadecimale e ASCII dei byte in memoria. In altre parole stampa il contenuto della memoria sotto

forma di numeri esadecimali seguiti dal corrispondente carattere ASCII.
9 - LOAD
Carica in memoria da nastro un programma ALPA.
10 - SAVE
Memorizza sul nastro il programma in memoria.
11 - NEW
Cancella il programma ALPA in memoria e tutti i suoi riferimenti.
12 - QUIT
Esce dall'ALPA per tornare al BASIC.

Tutti questi comandi possono essere abbreviati ricorrendo alle loro prime due lettere:

ENTER	si puo' scrivere	EN
MEMORY	" "	ME

NOTA: LIST puo' essere abbreviato con LI, ma LIST linea ### deve essere scritto per intero.

L'uso della memoria con ALPA

Avrete notato che finora abbiamo usato solo piccole aree di memoria per i programmi e i dati, non perche' siano le uniche utilizzabili, ma perche' siamo sicuri che queste non vengono usate ne' dal BASIC, ne' dal sistema operativo, ne' dall'ALPA.

In altre parole, il BASIC e il sistema operativo usano contemporaneamente delle aree di memoria per lavorare ed e' bene conoscere queste aree, per evitare che il sistema operativo o il BASIC si blocchino. (Ricordatevi che l'ALPA e' scritto in BASIC.) Se guardate le tabelle delle appendici 9 e 10, vedrete che esistono altre zone di memoria che possono essere utilizzate, oltre a quelle normalmente usate nel libro, che sono:

$C000 - $CFFF
$334 - $33B
$FB - $FE (pagina zero)

Le aree migliori da usare in pagina zero, quando questa e' piena, sono quelle riservate al buffer della cassetta ecc.

Se un programma non si ferma come dovrebbe, potete interromperlo premendo contemporaneamente i tasti RUN/STOP e RESTORE, passando il controllo al BASIC. Per ritornare

all'ALPA bisogna fare GOTO 2020. Questa e' la procedura da seguire quando si esce accidentalmente dall'ALPA e, se non funziona, bisogna rifare di nuovo RUN, perdendo il programma in L.M.

Quando viene chiesto l'indirizzo di partenza del programma in L.M. rispondete sempre $C000 a meno che non siate stati istruiti diversamente.

Adesso ripeteremo alcuni dei programmini precedentemente usati per vedere l'uso dell'ALPA. Esempio:

```
10 A901     LDA #$01
20 8D0004   STA $0400
30 60       RTS
```

Questo e' il programma usato all'inizio del capitolo 2. Usate l'ALPA per testare la locazione $400 (1024) prima e dopo il RUN del programma e per scrivere i numeri sul lato sinistro del programma sottostante:

```
10 A901
20 8D0004
30 60
```

Il calcolatore scrivera' il codice assembler sulla destra della linea. Per testare la variazione della locazione $400 fate:

```
WATCH        (Quale indirizzo)=$400
```

(se non avete definito l'indirizzo di partenza a $C000 fatelo ora con CHANGE). Battete ENTER, fate CLR per pulire lo schermo e poi date RUN. Adesso proviamo a modificare il programma, cambiando la linea:

```
A918     LDA #$18
Fate LIST.
Scrivete ENTER, pulite lo schermo con CLR e date
RUN.
```

Se vogliamo possiamo anche disassemblare il programma in memoria per essere sicuri che le variazioni siano state fatte correttamente con MEMORY. (Indirizzo C000). Per uscire dal disassemblato premete M e fate LIST. Volendo possiamo anche memorizzarlo sul nastro.

NOTA: SAVE e LOAD richiedono alcuni minuti per lavorare.

Adesso fate NEW e scrivete gli altri programmini presenti

nel capitolo 2, usando sempre l'ALPA e ricordando che questo usa sempre i numeri esadecimali al contrario di quello che abbiamo fatto nel capitolo 2.

Le altre caratteristiche e gli altri comandi verranno discussi piu' avanti quando sara' piu' rilevante il loro utilizzo.

Capitolo 4 SOMMARIO

1. Da questo capitolo in poi useremo l'ALPA per caricare in memoria tutti i programmi in L.M..
2. I comandi ALPA (che possono essere abbreviati alle prime due lettere) sono i seguenti:

LIST
ENTER
RUN
WATCH
NWATCH
CHANGE
MEMORY
DUMP
LOAD
SAVE
NEW
QUIT

3. Sebbene presenteremo tutti i programmi nella forma:

linea ### Istruzione in Esa Istruzione in Assembler, per digitare un programma sara'sufficiente scrivere la linea ### e l'esadecimale, perche' il calcolatore scrivera' automaticamente il codice assembler.

CAPITOLO 5

EQUIPAGGIAMENTO DEL MICROPROCESSORE

Nei capitoli precedenti abbiamo svolto una gran quantita' del lavoro di fondo necessario per comprendere gli intrecci della programmazione in linguaggio macchina; i restanti concetti di base saranno introdotti piu' avanti. A questo punto possiamo fare gia' alcune cose come ad esempio usare il codice macchina per fare dei calcoli.

Memorizzazione di numeri

Sappiamo dal capitolo 3 che il numero massimo memorizzabile in un singolo byte e' 255 e abbiamo anche visto che per indirizzamenti superiori a 255 occorrono due byte per rappresentarli nella forma byte basso - byte alto, cosicche' l'indirizzo = byte basso + 256 X byte alto.

Lo stesso metodo puo' essere usato per rappresentare ogni sorta di numero maggiore di 255 e minore di 65536 (65535 = 255+256X255) nonche' per numeri ancora piu' grandi

Num = 1° byte + 256 X 2° byte + 65536 X 3° byte + ...ecc.

Il flag di carry

Quando si sommano due numeri da un byte, puo' succedere che il risultato sia maggiore di 255. In questo caso cosa possiamo fare? Se mettessimo il risultato in un singolo byte, questo non sarebbe maggiore di 255. Cosi':

207 + 194 = 401 mod 255 = 145

ma anche

58 + 87 = 145

A questo punto e' ovvio che occorre trovare un sistema per memorizzare le informazioni extra che vengono perse quando il risultato e' maggiore di 255. Il microprocessore 6510 e' fornito di una struttura apposita, nella forma di

un singolo bit (dito) chiamato "flag di carry". Il flag di carry viene attivato se il risultato di un' operazione e' maggiore di 255 (funziona come il riporto). Esempio:

207 + 194 = 145; carry = 1
58 + 87 = 145; carry = 0

NOTA: un solo bit e'sufficiente per coprire tutti i casi di carry.

```
   11111111        255
 + 11111111       +255
                  ----
 1 11111110        254 + carry
 bit di carry
```

Percio', per sommare numeri da due byte, si sommano prima i due byte bassi, si memorizza il risultato e poi si sommano i byte alti includendo nella somma il bit di carry dell'addizione dei byte bassi. Esempio:

30A7 + 2CC4 = 5D6B

viene eseguita nella maniera seguente:

```
byte bassi
  A7
+ C4
  ---
  6B      carry = 1
  ---
 byte alti
  30
+ 2C
+  1      (bit di carry)
  ---
  5D
  ---
 Risposta = 5D6B
```

Addizione di numeri

L'istruzione in L.M. che somma due numeri da un byte e' ADC (add with carry - somma col carry), che aggiunge il numero specificato (o memoria) all'accumulatore e tiene il risultato nell'accumulatore stesso; l'istruzione somma automaticamente il bit di carry durante il calcolo. Percio' prima di qualsiasi addizione il carry deve essere messo a zero, a meno che questa operazione non abbia bisogno del carry di un'operazione precedente. Per mettere a zero il

carry si usa l'istruzione CLC (clear carry flag - metti a zero il carry) prima di ogni istruzione ADC.
Scrivete il seguente programma usando l'ALPA.

```
A quale indirizzo:? C000
10 A903            LDA #$03
20 18              CLC
30 6905            ADC #$05
40 8D3403          STA $0334
50 60              RTS

WATCH
Quale indirizzo? 334
ENTER
RUN
```

Il programma stampera':

```
'indirizzo 334 prima' = 00     3
'indirizzo 334 dopo'  = 08   + 5
                             ---
                             08
```

Adesso cambiamo le linee 10 e 30 per modificare la somma calcolata.

```
10 A927            LDA #$27
30 69F4            ADC #$F4
```

Fate ENTER e poi RUN ed il calcolatore rispondera':

```
'indirizzo 334 prima' = 08
'indirizzo 334 dopo ' = 1B
                                27
                              + F4
                             ----
carry attivato →            1 1B
```

NOTA: non si puo' dire che il carry e' stato attivato dal nostro risultato.

Adesso cambiamo di nuovo il programma e deliberatamente attiviamo il carry con l'istruzione SEC (set carry flag - metti a uno il carry) prima di qualsiasi addizione. Scrivete queste linee:

```
10 A903            LDA #$03
20 38              SEC
30 6905            ADC #$05
```

Fate ENTER e RUN ed il computer rispondera':

```
'indirizzo 334 prima' = 1B
'indirizzo 334 dopo'  = 09
```

```
    3
  + 5
  + 1        (bit di carry)
  ---
= 9
```

Scrivete queste linee:

```
10 A927          LDA #$27
15 18            CLC
20 69F4          ADC #$F4
25 A903          LDA #$03
30 6914          ADC #$14

ENTER e
RUN

'indirizzo 334 prima' = 09
'indirizzo 334 dopo'  = 18
```

Il carry non e' stato alterato fra la prima e la seconda addizione:

```
          27          3
        + F4        + 14
        ----        +  1 (carry)
Carry → 11B         ----
                  = 18
```

Adesso cambiate la linea 20 e ripetete

```
20 6920          ADC #$20

'indirizzo 334 prima' = 18
'indirizzo 334 dopo'  = 17
```

```
    27                 3
  + 20               + 14
  ----               +  0  (carry)
= 47  carry 0        ----
                   = 17
```

Da questi esempi vediamo come il bit di carry viene passato da un'addizione all'altra.

Useremo questi esempi per fare addizioni di numeri da due byte, usando il metodo precedentemente descritto.

Addizione di due byte

Supponiamo di voler sommare i numeri 6C67 e 49B2.

```
   6C67
 + 49B2
 =  ????
```

Per far cio' dobbiamo spezzare il problema in una addizione a un byte:

```
byte bassi   67          byte alti   6C
           + B2                    + 49
           1 19                    +  1 (carry)
           carry                     B6
```

Cancellate il vecchio programma con NEW e scrivete questo:

```
10 A967          LDA #$67
20 18            CLC
30 69B2          ADC #$B2
40 8D3403        STA $334
50 A96C          LDA #$6C
60 6949          ADC #$49
70 8D3503        STA $335
80 60            RTS
```

Esso memorizza il byte basso del risultato in 334 e il byte alto in 335. Per controllare la risposta usiamo il comando WATCH su entrambi i byte (facendo girare il programma due volte).

```
ENTER e

RUN

'indirizzo 334 prima' = ??
'indirizzo 334 dopo'  = 19
```

Adesso fate:

```
WATCH
Quale indirizzo? 335
RUN
```

```
'indirizzo 335 prima' = ??
'indirizzo 335 dopo'  = B6
```

Ora sommiamo il byte basso e quello alto del risultato per avere la risposta:

```
  6C67
+ 49B2
  B619
```

Sottrazione di numeri

Questa procedura puo' essere estesa per sommare numeri con qualsiasi lunghezza in byte.

Il microprocessore cosi' come possiede un comando per fare le addizioni, ne ha uno per le sottrazioni: nello stesso modo in cui il comando ADC usa il flag di carry nei calcoli, altrettanto fa l'istruzione SBC (subctract with carry - sottrai col carry). Per il modo in cui viene svolta la sottrazione il bit di carry viene invertito (1 diventa 0 e 0 diventa 1) nel calcolo

```
    8                  8
   - 5      but       - 5
   - 1                - CARRY   (CARRY = 1)
=    2            =     3
```

Conseguentemente, per fare una sottrazione senza il carry, il suo flag deve essere messo a uno, prima che venga usato il comando SBC. Scrivete:

```
NEW per cancellare il programma
10 A908            LDA #$08
20 18              CLC
30 E905            SBC #$05
408D3403           STA $334
50 60              RTS
WATCH
Quale indirizzo? 334
ENTER
RUN
```

Mettere a zero il carry invece di metterlo a uno, produce una risposta sbagliata; correggiamo l'errore mettendo il

carry a uno prima della sottrazione.

```
20 38              SBC
ENTER
RUN
```

Ora la risposta e' corretta:

```
    8                 8
  - 5               - 5
  - 1 (CARRY=0)     - 0 (CARRY=1)
= 2               = 3
```

Adesso vi chiederete come il microprocessore esegua le sottrazioni con il risultato minore di zero. Per esempio 8-E = -6. Cambiate la linea 30, poi fate ENTER e RUN.

```
30 E90E            SBC #$0E

'indirizzo 334 prima' = ??
'indirizzo 334 dopo'  = FA
```

```
    8   oppure   BORROW (prestito) = 108  carry
  - E                                - E  a zero
  - 6                                FA
```

NOTA: -6 = 0-6 = FA
FA+6 = 0

La messa a zero del carry per simulare il prestito di una cifra viene usata tanto per sottrazioni di piu' byte,quanto per addizioni dello stesso tipo. **Scrivete un programma che** faccia questa sottrazione:

E615 - 7198

ecco un esempio:

```
NEW
10 A915            LDA #$15
20 38              SEC
30 E998            SBC #$98
40 8D3403          STA $334
50 A9E6            LDA #$E6
60 E971            SBC #$71
70 8D3503          STA $335
80 60              RTS

ENTER
RUN
```

Annotate il risultato e usate WATCH per vedere la cella 335, che e' il byte alto del risultato; poi fate di nuovo RUN. Sommate il byte alto con quello basso del risultato per avere la risposta 747D.

Queste istruzioni ADC e SBC si possono usare in molti modi di indirizzamento, come molte altre istruzioni. In questo capitolo abbiamo usato solo l'indirizzamento immediato.

NOTA: SEC CLC hanno un solo modo di indirizzamento (implicito). Esse svolgono una funzione specifica all'interno di un dato registro e non vi sono alternative al loro indirizzamento; il loro metodo di indirizzamento e' "implicito" (stabilito) all'istruzione.

Un esercizio

Scrivete un programma che sommi il valore $37 al contenuto della locazione di memoria $334 usando ADC col metodo di indirizzamento "assoluto". Usate WATCH per osservare il risultato.

NOTA:

```
LDA #$FF
CLC
ADC #$01
```

lascia il valore $0 in A e il carry a uno, e

```
LDA #$00
SEC
SBC #$01
```

lascia il valore $FF in A e il carry a zero (prestito).

Percio' abbiamo quello che si dice "wraparound" (avvolgimento su se stesso). Contando oltre 255 riparte da zero e contando al di sotto dello zero continua da 255 in giu'.

Capitolo 5 SOMMARIO

1. Ogni tipo di numero si rappresenta usando piu' di un byte.
Num = 1° byte + 2° byte X 256 + 3° byte X 65536 + ...ecc.
2. Il microprocessore 6510 ha il flag di carry per segnalare un riporto nel byte alto in un'addizione di due byte.

3. ADC somma due byte piu' il contenuto del flag di carry. Viene usata CLC se il carry e' ininfluente sul risultato dell'addizione.
4. ADC mette a uno il flag di carry se il risultato e' piu' grande di 255, e lo mette a zero se non lo e'. La risposta trovata nell'accumulatore e' sempre minore di 256. (A = Risultato mod 256)
5. SBC sottrae la memoria dall'accumulatore e poi sottrae l'inverso del carry. Per far si' che il carry non interferisca con i calcoli bisogna fare SEC prima di un'istruzione SBC.
6. SBC mette a uno il carry se il risultato non richiede un prestito (A-M >= 0). Il carry viene messo a zero se (A-M < 0) e il risultato che rimane in A e' 256-(A-M).
7. Addizione di due byte:

```
CARRY A ZERO
XX = SOMMA BYTE BASSI + (CARRY = 0)
YY = SOMMA BYTE ALTI + (CARRY = ?)
Risultato = $YYXX
```

8. Sottrazione di due byte:

```
CARRY A UNO
XX = SOTTRAZIONE BYTE BASSI - INVERSO (CARRY = 1)
YY = SOTTRAZIONE BYTE ALTI - INVERSO (CARRY = ?)
Risultato = $YYXX
```

CAPITOLO 6

CONTROLLO DEL PROGRAMMA

Sprite

Nel capitolo 2 abbiamo visto come si visualizzano le informazioni sul video mettendo dei dati nella "memoria di schermo". C'e' sul C64 un chip speciale che ha il compito di orientare il video ed e' chiamato "VIC II". (Una breve guida si trova nell'appendice 6). Usando le tecniche di addizione e sottrazione imparate nel capitolo precedente, vedremo alcune funzioni particolari adattabili con questo chip. Battete il seguente programma:

```
NEW
NWATCH
10 A901          LDA #$01
20 8D15D0        STA $D015
30 8D27D0        STA $D027
40 8D4020        STA $2040
50 A932          LDA #$32
60 8D00D0        STA $D000
70 8D01D0        STA $D001
80 60            RTS
ENTER e RUN.
```

Esso crea un quadrato in sovraimpressione sull'angolo sinistro alto dello schermo. Questo quadrato e' noto come "sprite": formato da nove caratteri normali puo' muoversi sullo schermo con facilita', velocemente e sopra gli altri caratteri. E' controllato dai registri del chip "VIC II", che sono simili a quelli del microprocessore ma, in conformita' al loro uso, sono "mappati" direttamente in memoria agli indirizzi da D000 a D02E. Il termine "mappati" sta' a significare che questi byte sono messi sopra la memoria. Quando si accede a questa zona di memoria, si opera con i registri del "VIC II" o con qualunque cosa mappata in quella zona. Per usare la similitudine dell'ufficio postale, immaginate questa sorta di memoria mappata come un gruppo di caselle postali senza fondo che sono collegate tramite piani inclinati con macchine speciali poste in altra parte dell'ufficio.

Nota della scansione: si usa STA $2040 anziché STA 2040 (o STA $07F8)

Movimento dello sprite

Adesso vedremo di scrivere un programma che muova sprite sullo schermo.

I due registri che controllano la posizione degli sprite (coordinate X e Y) sono locati agli indirizzi D000 e D001. Il nostro programma aggiunge piu' volte 4 al valore della coordinata X della locazione D000, facendo muovere orizzontalmente lo sprite. Inserite queste linee sul programma scritto prima:

```
 80 AD00D0          LDA #$D000
 90 6904            ADC #$04
100 8D00D0          STA $D000
110 6904            ADC #$04
120 8D00D0          STA $D000
130 6904            ADC #$04
140 8D00D0          STA $D000
150 6904            ADC #$04
160 8D00D0          STA $D000
170 6904            ADC #$04
180 8D00D0          STA $D000
190 60              RTS
ENTER e RUN.
```

Potete vedere che lo sprite si muove sullo schermo: e' probabile che i movimenti siano cosi' veloci che non riusciate a vederli in dettaglio, ma la cosa certa e' che lo sprite si muove. Nonostante queste numerose istruzioni in L.M., noterete che lo sprite non si e' spostato di molto e molto a lungo. Come possiamo fare per spostare lo sprite su tutto lo schermo e farlo muovere per sempre? E' necessario aggiungere molte istruzioni in L.M.? Quello di cui abbiamo bisogno e' un comando come "GOTO" del BASIC. E' necessario mettere le istruzioni che cambiano le coordinate in un ciclo che ripeta piu' volte le stesse cose.

Cicli con l'uso di JMP

Per l'esecuzione di cicli c'e' l'istruzione(JUMP-SALTA). Come per il "GOTO" del BASIC dobbiamo dire a JMP dove saltare, nella forma JMP indirizzo (JMP byte basso - byte alto) (Indirizzamento assoluto).

Usiamo JMP per scrivere un programma in L.M. equivalente

a questo programma BASIC.

```
· INIZIALIZZA
·
·

100 X=X+4
110 GOTO 100
·
·
·
```

Scrivete:

```
190 )
180 |
170 |
160 }     per cancellare queste linee
150 |
140 |
130 |
120 )
110 4CL90        JMP L90
120 60           RTS
```

Indirizzamento con i numeri di linea in ALPA.

Il metodo di indirizzamento usato alla linea 110 e' un tipo particolare dell'ALPA. Un vantaggio dell'ALPA e' quello di calcolare gli indirizzi per voi. Normalmente, usando l'istruzione JMP col metodo di indirizzamento assoluto, bisogna ricavare l'indirizzo a cui JMP deve andare; cio' puo' provocare degli errori o dei nonsensi, come mostrato in questi esempi:

```
1.C000:4C08C0      JMP $C008
  C003:A902        LDA #$02
  C005:8D3403      STA $0334
  C008:60          RTS

2. 335:4C3D03      JMP $33D
   338:A902        LDA #$02
   33A:8D3403      STA $334
   330:60          RTS

3.C000:4C0BC0      JMP $C00B
  C003:A902        LDA #$02
  C005:18          CLC
  C006:6904        ADC #$04
  C008:8D3403      STA $334
  C00B:60          RTS
```

Creare un secondo programma dal primo

In altre parole, per muovere lo stesso programma fra differenti parti di memoria, dovreste scorrere l'intero programma, cambiare ogni volta gli indirizzi di ogni JMP all'interno dello stesso e farli puntare al loro nuovo indirizzo.

Creare un terzo programma dal primo

Viene fatto con l'aggiunta di semplici comandi, cosa che capita spesso durante la messa a punto di programmi. Potreste dover cambiare l'indirizzo di ogni JMP, operazione estremamente frustrante, di notevole perdita di tempo e facilmente soggetta a errori. L'ALPA ha la possibilita' di fare JMP a numeri di linea (numero di linea del sorgente). Quando viene dato ENTER, l'ALPA converte questo riferimento a un numero di linea in un indirizzo assoluto rendendolo comprensibile al microprocessore.

Battete LIST, per vedere il listato del programma scritto in precedenza.

```
 10 A901          LDA #$01
 20 8D15D0        STA $D015
 30 8D27D0        STA $D027
 40 8D4020        STA $2040
 50 A932          LDA #$32
 60 8D00D0        STA $D000
 70 8D01D0        STA $D001
 80 AD00D0        LDA $D000
 90 6904          ADC #$04
100 8D00D0        STA $D000
110 4CL90         JMP L90
120 60            RTS
```

e poi ENTER.

Il computer risponde con:

```
110 4CL90
 90 6904
OK
```

che indica che il calcolo dell'indirizzo della linea 90

Nota della scansione: si usa STA $2040 anziché STA 2040 (o STA $07F8)

(L90) e' stato fatto. Fate MEM per vedere come e' messo in memoria il listato del programma:

Disassemblo a partire da? C000

Il calcolatore risponde con:

```
C000:A901          LDA #$01
C002:8D15D0        STA $D015
C005:8D27D0        STA $D027
C008:8D4020        STA $2040
C00B:A932          LDA #$32
C00D:8D00D0        STA $D000
C010:8D01D0        STA $D001
C013:AD00D0        LDA $D000
C016:6904          ADC #$04
C018:8D00D0        STA $D000
C01B:4C16C0        JMP $C016
C01E:60            RTS
C01F:? .           ?
  .    .           .
  .    .           .
```

Come potete vedere, l'ALPA ha calcolato il corretto indirizzo.

Adesso usate il comando CHANGE per mettere il programma alla locazione $334, fate ENTER e disassemblate. Alle locazioni $34F e $351 si nota che l'ALPA ha di nuovo cambiato l'indirizzo del salto correttamente, senza nessun intervento da parte vostra. Ora con CHANGE riportate il programma all'indirizzo $C000, fate ENTER e poi RUN.

Lo sprite si muove sullo schermo cosi' velocemente da sembrare sfumato: una piccola dimostrazione della velocita' di un programma in codice macchina.

Cicli infiniti

Bisogna notare un'altra cosa e cioe' che il programma non si e' ancora fermato. Come il programma:

```
.
.
100 X=X+4
110 GOTO 100
.
.
```


Nota della scansione: si usa STA $2040 anziché STA 2040 (o STA $07F8)

il nostro continua a girare all'interno del ciclo; in questo caso si dice che il programma e' entrato in un "ciclo infinito".

Il tasto RUN/STOP non ci fa uscire dal ciclo, come si verificherebbe se il programma fosse in BASIC; quest'ultimo infatti ha una parte in L.M. che controlla sempre se il tasto RUN/STOP e' stato premuto, ma noi questa parte non l'abbiamo messa nel nostro programma in L.M. Esistono nel nostro caso due soli modi per uscire dal ciclo infinito: uno e' premere contemporaneamente i tasti RUN/STOP e RESTORE; cio' provoca un segnale hardware -NMI (Non Maskable Interrupt - Interruzione non mascherabile) che interrompe il calcolatore e lo fa tornare al BASIC. L'altro modo e' di spegnere il computer. Premete RUN/STOP e RESTORE per fermare il programma; adesso siete in BASIC, ma per continuare con il vostro programma in ALPA fate:

```
GOTO 2020
```

Non esiste altro modo per uscire da una routine in L.M. a meno che non ritorni da sola con un'istruzione RTS. Fate LIST. NOTA: a causa del JMP il programma non va mai all'istruzione RTS, come questo programma BASIC:

```
 5 X=4
10 PRINT"HELLO";X
15 X=X+4
20 GOTO 10
30 END
```

Il programma non termina a causa del GOTO in linea 20. Per far scrivere allo stesso programma HELLO 4 fino a HELLO 100 dobbiamo scrivere:

```
 5 X=4
10 PRINT"HELLO";X
15 X=X+4
20 IF X=104 GOTO 40
30 GOTO 10
40 END
```

In questo caso la linea 20 andra' alla linea 40 solo se X=104, facendo terminare il programma all'istruzione END. Se X non e' uguale a 104, il programma va alla linea 30 e continua il suo ciclo a linea 10. Per fare la stessa cosa in L.M. ci occorre un'istruzione che faccia il confronto fra due numeri (X e 1049) e un'altra che faccia "saltare"

il programma in base al risultato del confronto (IF...GOTO40).

Confronto di numeri

In precedenza abbiamo incontrato (capitolo 5) l'idea di "flag", che non e' altro che il valore di un singolo bit all'interno del microprocessore. Nello stesso capitolo 5 abbiamo pure incontrato il flag di carry, che viene usato per segnalare la necessita' di un riporto in un'addizione di piu' byte (o un prestito in una sottrazione anch'essa multibyte). Il microprocessore ha sette flag, per differenti scopi, che tiene in un registro dalle funzioni speciali chiamato Processor Status Code Register (Registro dei codici dello stato dei processi) o Status Byte (byte di stato). Questi sette flag, di cui uno e' vuoto, sono rappresentati dal proprio bit all'interno di questo byte ed occorrono comandi speciali per operare con loro. Detti flag sono attivati o meno da molte istruzioni in L.M. (per saperne di piu' guardate il capitolo 10). Per esempio: ADC attiva o meno il flag di carry a seconda del risultato di un'addizione; allo stesso modo 'CMP' (Compare - confronta), che confronta il contenuto dell'accumulatore con il contenuto della memoria (dipende anche dal metodo di indirizzamento), influenza col suo risultato il byte di stato.

Istruzioni "Branch"

L'altra istruzione di cui abbiamo bisogno e' quella che fa saltare ad un indirizzo, a seconda del valore di uno dei flag di stato. La forma di quest'istruzione viene chiamata istruzione di "branch" (salto). E' differente dall'istruzione JMP (pur avendo lo stesso significato letterale) non solo per il fatto che e' condizionata (dipende dalla condizione dei registri di stato), ma anche perche' e' l'unica ad usare il metodo di indirizzamento relativo, che sta a significare che l'indirizzo e' calcolato relativamente all'istruzione di salto. Dettagli maggiori sull'indirizzamento relativo li troverete alla fine del capitolo; nel frattempo useremo l'ALPA per calcolare gli indirizzi come abbiamo fatto per l'istruzione JMP.

Flag di zero

Per sapere il risultato di un'istruzione CMP, e cioe' se due numeri confrontati sono uguali, useremo il comando BEQ (Branch on equal - Salta se uguale).

Aggiungete queste linee al nostro programma:

```
 90 6901          ADC #$01
110 C964          CMP #$64
120 F0L140        BEQ L140
130 4CL90         JMP L90
140 60            RTS
```

La linea 90 e' stata modificata per far si' che lo sprite non si sposti troppo velocemente. Fate LIST, ENTER e poi RUN.

NOTA: L'ALPA ha calcolato entrambi gli indirizzi usando il numero di linea.

Lo sprite si e' mosso fino a meta' schermo per poi fermarsi normalmente con l'istruzione RTS.

Riepilogo del programma

```
linee  10 -  80 inizializzazione
linee  90 - 130 ciclo di movimento dello sprite
linee 110 - 120 test per la condizione di fine
linea 140       FINE
```

Abbiamo lavorato a cercare un metodo per usare un ciclo che testi una condizione con cui uscire dal ciclo. Possiamo pero' fare un programma piu' efficiente, che cicli finche' una certa condizione si e' verificata. La differenza e' chiara, ma diventa lampante confrontando questo programma BASIC con il precedente.

```
 5 X=4
10 PRINT"HELLO";X
15 X=X+4
20 IF X<>104 GOTO 10
30 END
```

In questo modo si risparmia una linea di programma, il che diventera' utile quando lo spazio occupato e la velocita' saranno importanti per un programma. Ad ogni modo e' sempre buona norma scrivere i programmi in codice tenendo a mente queste considerazioni, per ottenere dei prodotti finali

piu' corti, facili da leggere e da mettere a punto.

Questa tecnica di programmazione si traduce in L.M. con il comando BNE (Branch not equal - Salta se diverso). Per provarlo fate:

```
120 D0L90          BNE L90
130
```

e poi fate il LIST del programma cosi' com'e'.

Riepilogo del programma

```
linee  10 -  80 inizializzazione
linee  90 - 120 ciclo di movimento dello sprite
linea 140       FINE
```

Come potete vedere, cambiando il ciclo abbiamo districato il flusso del programma. Fate ENTER e RUN, per verificare che il programma faccia le stesse funzioni nonostante i cambiamenti. Riepilogando: i modi di scrivere un programma sono molti (giusti o sbagliati che siano non lo possiamo dire), ma il miglior programma e' quello che e' piu' leggibile e piu' facile da mettere a punto.

Ci sono un sacco di cose che possiamo imparare conoscendo il modo in cui un'istruzione lavora. Per esempio l'istruzione CMP effettua il confronto fra due numeri facendo una sottrazione (accumulatore - memoria) senza memorizzare il risultato, attivando o meno il flag di stato. Le istruzioni che abbiamo appena usato (BEQ e BNE) non si riferiscono all'eguaglianza fra i due numeri confrontati, ma di fatto testano il flag di 'zero' dei registri di stato:

```
BEQ - Branch on equal to zero
      (Salta se uguale a zero)
BNE - Branch on not equal to zero
      (Salta se diverso da zero)
```

E' la condizione del flag di zero che viene attivata dal risultato della sottrazione fatta dal comando CMP (accumulatore - memoria = 0 che mette il flag di zero = 1). Questo flag viene in seguito testato dai comandi BEQ e BNE. Tutto cio' puo' sembrare senza senso finche' non vi rendete conto che, poiche' il comando CMP fa una sottrazione, il flag di carry puo' essere influenzato dal risultato. In altre parole, se la sottrazione svolta da CMP necessita di

un "prestito" (borrow) (A - Mem < 0, A minore della memoria), allora il flag di carry viene disattivato (CARRY = 0). Se la sottrazione non necessita di un prestito (A - Mem >= 0, A maggiore o uguale della memoria), allora il carry e' attivato (CARRY = 1).

Quindi il comando CMP testa non solo se A = Mem ma anche se A < Mem e anche A >= Mem; percio' (se A >= Mem ma A <> Mem) allora A > Mem. Adesso possiamo scrivere il nostro programma BASIC:

```
 5 X=4
10 PRINT"HELLO";X
15 X=X+4
20 IF X<101 GOTO 10
30 END
```

Cio' rende il programma piu' auto-esplicativo, mostrando chiaramente che se il valore di X e' maggiore di 100, non viene stampato. Per vedere se l'accumulatore e' minore della memoria usate il comando BCC (Branch on carry clear - Salta se il carry e' a zero) al fine di verificare se per caso avete avuto un prestito. Per sapere se l'accumulatore e' maggiore o uguale alla memoria usate il comando CMP seguito da BCS (Branch on carry set - Salta se il carry e' a 1).

Scrivete un programma in L.M. per muovere lo sprite sullo schermo usando il test di A < Memoria (come nel precedente programma BASIC).

Indirizzamento relativo

Tutte le istruzioni di salto (branch) usano questo tipo di indirizzamento (JMP non e' un'istruzione di tipo branch). Nell'indirizzamento relativo, l'indirizzo (la destinazione del salto) viene calcolato relativamente all'istruzione. Tutte queste istruzioni sono di due byte: un byte per l'istruzione e l'altro per specificare in vari modi l'indirizzo. Esse operano col secondo byte, specificando una deviazione al prossimo byte dopo l'istruzione, in accordo con la tabella dell'appendice 4. Da 0 a 7F significa un equivalente salto in avanti e da 80 a FF un salto indietro di 256 byte.

```
        F0 03              BEQ laggiu'
        8D 34 03           STA $334
laggiu' 60                 RTS
```

Il valore 3, come parte dell'istruzione, e' il numero di byte dall'inizio della prossima istruzione (8D).

primo prossimo byte (34)
secondo prossimo byte (03)
terzo prossimo byte (60)

Con il successivo programma controlliamo che l'indirizzo di arrivo del salto sia di fatto l'indirizzo dell'istruzione dopo il salto, piu' il valore fisso.

```
C000:F003          BEQ $C005
C002:8D3403        STA $0334
C005:60            RTS
```

e

```
334:F003           BEQ $0339
336:8D3403         STA $0334
339:60             RTS
```

Il codice macchina rimane lo stesso ma il disassemblato e' diverso. Il programma lavora allo stesso modo per qualsiasi indirizzo; cio' e' esattamente l'opposto di quello che succede con JMP che usa l'indirizzamento assoluto e non puo' essere "rilocato" (spostato ad un altro indirizzo di memoria). Fortunatamente non dobbiamo ricalcolare il valore fisso ogni volta che inseriamo un'istruzione fra l'istruzione di salto e la destinazione. Usando l'ALPA possiamo evitare di calcolare il valore fisso dando il comando di salto con L, numero di linea.

Scrivete con ALPA alcuni programmi che usino istruzioni di salto con il numero di linea, controllate il disassemblato del codice inserito e che il salto fatto sia corretto, confrontandolo con la tabella dei salti relativi nell'appendice 4.

Capitolo 6 SOMMARIO

1. Uno sprite e' un quadrato di nove caratteri normali (3X3) che si puo' muovere in sovraimpressione sugli altri caratteri dello schermo, cambiando le sue coordinate X e Y contenute in uno dei registri del chip "VIC II".
2. Il comando JMP indirizzo e' l'equivalente del GOTO numero di linea del BASIC. Fa si' che il programma salti all'indirizzo specificato.

3. L'ALPA puo' far riferimento ad indirizzi interni al programma utilizzando il loro numero di linea. Esempio:

JMP L40 (Salta alla linea 40)

4. Per uscire da un ciclo infinito premere contemporaneamente i tasti RUN/STOP e RESTORE. Per far ripartire l'ALPA senza perdere il programma in L.M., fate GOTO 2020.
5. Il registro di stato del microprocessore ha sette flag (uno vuoto) che possono essere attivati da molte istruzioni in codice macchina.
6. Le istruzioni "branch" effettuano salti condizionati allo stato del flag a cui si riferiscono. Esempio:

BEQ	Branch on equal	Z=1
BNE	Branch on not equal	Z=0
BCS	Branch on carry set	C=1
BCC	Branch on carry clear	C=0

7. L'istruzione CMP confronta due byte (facendo una sottrazione senza memorizzare il risultato). Solo i flag sono interessati da questa operazione.

FLAG	CARRY	ZERO	SIGNIFICATO
	0	0	A < Mem
	?	1	A = Mem
Valore	1	?	A >= Mem
	1	0	A > Mem

8. Il metodo di indirizzamento relativo, utilizzabile solo per le istruzioni "branch", specifica un indirizzo relativo all'istruzione che usa. Esempio: BNE 03 significa che bisogna saltare i prossimi tre indirizzi di memoria (vedere tabella nell'appendice 4).
9. L'ALPA calcola questi indirizzi automaticamente purchè si specifichi il salto a L, dove L e' il numero di linea.

CAPITOLO 7

CONTATORI, CICLI, E PUNTATORI

Contatori per controllare un ciclo

Supponiamo di dover moltiplicare due numeri e, poiche' non c'e' nessuna istruzione in L.M. che faccia questa funzione, dobbiamo costruire un programma apposta. Sommeremo un numero al totale tante volte quanto e' grande l'altro numero. Esempio:

```
10 A=7:B=3
20 T=T+A
30 T=T+A
40 T=T+A
50 PRINT"7X3=";T
```

Sarebbe piu' facile e pratico (specialmente per numeri grandi) usare questo ciclo:

```
10 A=7:B=3
20 T=T+A
30 B=B-1
40 IF B<>0 GOTO 20
50 PRINT"7X3=";T
```

NOTA: questo non vuole essere il metodo migliore per moltiplicare due numeri, ma a noi interessano solo le istruzioni usate. Il metodo migliore e' descritto nel capitolo 10.

Contatori che utilizzano l'accumulatore

In questo piccolo programma ci sono due variabili: A, che viene sommata al totale e B, che controlla il ciclo. In questo caso non potremmo fermare il programma controllando il totale, come abbiamo fatto prima, perche' non potremmo conoscere il totale prima di aver scritto il programma

stesso. Il programma in L.M. sulla traccia di quello fatto precedentemente sarebbe:

```
        LDA #$00
        STA A
        LDA #$03
        STA B
ciclo   LDA A
        CLC
        ADC #$07
        STA A
        LDA B
        SEC
        SBC #$01
        STA B
        BNE ciclo
        RTS
```

Contatori che utilizzano la memoria

La maggior parte di questo programma consiste di istruzioni che caricano e scaricano fra accumulatore e memoria. Come ci e' capitato per altre istruzioni, tutte queste operazioni sono svolte da comandi speciali. INC (Increment memory - Incrementa la memoria) somma 1 al contenuto dell'indirizzo specificato e rimette il risultato nella stessa locazione; lo stesso accade per DEC (Decrement memory - Decrementa la memoria), il quale invece sottrae 1 dalla memoria.

NOTA: DEC e INC non attivano il flag di carry, ma quello di zero.

Ora possiamo scrivere il nostro programma in questo modo:

```
10 A903          LDA #$03
20 8D3403        STA $334
30 A900          LDA #$00
40 18            CLC
50 6907          ADC #$07
60 CE3403        DEC $334
70 D0L40         BNE L40
80 8D3503        STA $335
90 60            RTS
WATCH  indirizzo? 335
ENTER
RUN
```

Riepilogo del programma

```
linee 10 - 30 Inizializzazione
linee 40 - 70 Cicla finche' il risultato di DEC=0
linee 80 - 90 Fine
```

Utilizzando INC e DEC, possiamo usare qualsiasi locazione di memoria come contatore, lasciando libero l'accumulatore per altre cose.

Registri X e Y

Riscrivete il programma precedente usando INC e CMP per controllare la fine del ciclo.

I registri X e Y

Esiste un altro metodo piu' semplice per avere dei contatori senza usare INC e DEC. Nel capitolo 2 abbiamo detto che il microprocessore 6510 ha tre registri (A, X e Y) atti a molti scopi; negli ultimi cinque capitoli abbiamo parlato esclusivamente del registro A, l'accumulatore. A questo punto sembrera' ovvia la domanda: a cosa servono gli altri due registri?

E che cosa significa "atti a molti scopi"? Indietro abbiamo visto un registro con un solo scopo, come il byte di stato (ne incontreremo altri due piu' avanti), che viene usato solo per contenere i flag di stato e nient'altro, paragonato all'accumulatore che puo' contenere numeri fino a 255. I registri X e Y possono, come l'accumulatore, contenere numeri fino a 255 ma, al contrario di questo, non possono fare molte altre operazioni, come ad esempio addizioni e sottrazioni; pero' sono molto adatti ad essere usati come contatori.

Essi possono fare queste operazioni (confrontate con quelle dell'accumulatore e della memoria).

LDA LOAD ACCUMULATOR WITH MEMORY
LDX LOAD X WITH MEMORY
LDY LOAD Y WITH MEMORY

STA STORE ACCUMULATOR TO MEMORY
STX STORE X TO MEMORY
STY STORE Y TO MEMORY

INC INCREMENT MEMORY
INX INCREMENT X
INY INCREMENT Y
] IMPLIED ADDRESSING MODE

DEC DECREMENT MEMORY
DEX DECREMENT X
DEY DECREMENT Y
] IMPLIED ADDRESSING MODE

CMP COMPARE ACCUMULATOR WITH MEMORY
CPX COMPARE X WITH MEMORY
CPY COMPARE Y WITH MEMORY

Il registro X come contatore

Riscriviamo il programma usando il registro X come contatore:

```
NEW
WATCH                (What address?) 335
10 A203              LDX #$03
20 A900              LDA #$00
30 18                CLC
40 6907              ADC #$07
50 CA                DEX
60 D0L30             BNE L30
70 8D3503            STA $335
80 60                RTS
ENTER
RUN
```

Questa routine e' leggermente piu' corta e sensibilmente piu' veloce dell'originale, ma per il resto non ci sono differenze. Riscrivete i programmi che adoperano il registro X e rimpiazzateli con l'equivalente comando del registro Y. Fate pratica dell'uso dei registri X e Y, sostituendoli dove possibile all'accumulatore nei programmi descritti ai capitoli precedenti.

Spostamenti di blocchi di memoria

Come possiamo scrivere un programma che sposti da un posto all'altro un blocco di memoria? Ad esempio, come spostare la memoria da C100 - C150 a C200 - C250? Ovviamente si potrebbe anche cosi':

```
LDA $C100
STA $C200
LDA $C101
STA $C201
LDA $C102
    .
    .
    .
ecc.
```

Ma sarebbe il programma piu' ridicolo che si possa scrivere e non ne varrebbe proprio la pena data la sua grandezza.
Potremmo invece scrivere il programma:

```
LDA $C100
STA $C200
```

seguito da alcuni codici che incrementino la parte dell'indirizzo dell'istruzione. Questo concetto e' molto interessante e deve essere tenuto in considerazione. Il programma in questione si modifica mentre gira e viene chiamato "codice automodificato". Ma, per il fatto che modifica se stesso da solo, viene considerato una pratica molto poco usata perche' puo' provocare errori di proporzioni catastrofiche. E inoltre non e' molto facile da mettere a punto; rimane un concetto molto interessante, ma da non usare in programmi seri. A questo punto e' ovvio che anche questo metodo non risolve i nostri problemi.

La risposta al nostro problema sta' nei metodi di indirizzamento. Originariamente noi li abbiamo identificati come le vie di accesso ai dati e alla memoria in modi diversi e in forme diverse. Precedentemente abbiamo visto che:

Indirizzamento implicito

Il dato e' specificato come parte dell'istruzione. Esempio SEC, DEY.

Indirizzamento relativo

Indirizzo relativo all'istruzione, usato solo con i salti di tipo "branch".

Indirizzamento assoluto

Il dato e' specificato dall'indirizzo in due byte nel formato byte basso - byte alto.

Indirizzamento in pagina zero

Il dato e' specificato dall'indirizzo in un byte, contenuto nei primi 256 byte di memoria.

Indirizzamento indicizzato

E' per noi un nuovo metodo di indirizzamento e trova il dato da usare nell'istruzione sommando un byte indice

all'indirizzo assoluto definito nell'istruzione stessa. Il byte indice viene preso dai due registri X o Y (a seconda dell'istruzione usata). Per questo motivo i registri X e Y sono chiamati "registri indice".

Ricorrendo all'analogia con l'ufficio postale, e' come se ci avessero dato due pezzi di carta, uno con un indirizzo da due byte e l'altro da uno (0 - 255), che dobbiamo sommare insieme per ottenere il corretto indirizzo della casella.

Il registro X come indice

Con questo metodo di indirizzamento il programma che sposta un blocco di dati in un'altra zona di memoria diventa molto semplice.

```
NEW
10 A200          LDX #$00
20 BDC804        LDA $04C8,X
30 9DF004        STA $04F0,X
40 E8            INX
50 E028          CPX #$28
60 D0L20         BNE L20
70 60            RTS
ENTER
```

NOTA: la forma mnemonica dell'indirizzamento indicizzato e' composta da: l'indirizzo assoluto, una virgola e dal registro indice usato.

```
BDC804           LDA $04C8,X
B9C804           LDA $04C8,Y
```

L'istruzione cambia a seconda del registro. Fate RUN e, come vedrete, abbiamo usato di nuovo la memoria di schermo per verificare se abbiamo effettivamente duplicato una parte della memoria. Una linea di schermo e' stata copiata sulla successiva (la sesta sulla settima e cosi' via). Se non siete convinti fate un test sulla sesta linea per verificarne l'effetto.

La non-simmetria dei comandi

Se, come avevamo detto introducendo i registri X e Y, avete provato a sostituire A con i registri X e Y nei programmi

precedenti, starete chiedendovi se e' possibile farlo anche qui. La risposta e' no. Non tutti i comandi possono usare tutti i metodi di indirizzamento; nemmeno X e Y (X e' ovvio) possono usare il metodo di indirizzamento con l'indice X come e' stato fatto per STA. (E' possibile fare un LDA INDIRIZZO,X ma non STY INDIRIZZO,X). Per sapere tutti i metodi di indirizzamento per ogni istruzione date un'occhiata all'appendice 1.

Ricerca attraverso la memoria

Possiamo usare le conoscenze acquisite fino a questo punto, per fare alcuni esercizi abbastanza semplici. Se vi chiedessero di trovare la quarta occorrenza di un numero, ad esempio A9 all'interno di 255 byte a partire da un dato indirizzo, come opererereste?

Il modo migliore e' lavorare semplicemente e secondo la vostra logica. Per trovare la prima occorrenza di A9 da F000 in poi si puo' scrivere:

```
                LDY #$00
                LDA #$A9
ciclo           CMP $F000,Y
                BEQ trovato
                INY
                BNE ciclo
                RTS (Non ho trovato A9
                     fra F000 e F0FF)
trovato         RTS (Ho trovato un A9)
```

Potremmo mettere un programma che serve da contatore all'esterno della routine:

```
                LDX #$00
contaciclo      TROVATO "A9"
                INX
                CPX #$04
                BNE contaciclo
```

E combinando queste cose insieme otteniamo un programma unico:

```
10 A200         LDX #$00
20 A000         LDY #$00
```

```
 30 A9A9          LDA #$A9
 40 D900F0        CMP $F000,Y
 50 F0L90         BEQ L90
 60 C8            INY
 70 D0L40         BNE L40
 75 8E3403        STX $0334
 80 60            RTS
 90 E8            INX
100 E004          CPX #$04
110 D0L60         BNE L60
115 8E3403        STX $0334
120 60            RTS
```

Nel programma - quando e' finito, se X=4, la quarta occorrenza si trova in F000 + Y (esce tramite RTS in linea 120),
- se X < 4, non ci sono 4 occorrenze di A9 fra F000 e F0FF (esce tramite RTS in linea 80),
- linea 110 continua la routine di ricerca per "INY".

Fate WATCH indirizzo? 334
ENTER e RUN.

I risultati vi diranno se sono state trovate le 4 occorrenze di "A9". Cambiate il programma per sapere dove e' stata trovata la quarta occorrenza (STY $334). Fate ENTER e RUN per vedere i risultati. Verificate se questi sono corretti con il comando DUMP, che visualizza la memoria (a partire dall'indirizzo F000).

Adesso cambieremo alcune cose per districare un po' il programma. Scrivete queste linee:

```
 40 C8            INY
 50 F0L110        BEQ L110
 60 D9FFEF        CMP $EFFF,Y
 70 D0L40         BNE L40
 80 E8            INX
 90 E004          CPX #$04
100 D0L40         BNE L40
110 8E3403        STX $0334
120 60            RTS
 75
115
```

che rendono il programma meno intricato e piu' facile da

seguire. LIST

```
linee  10 -  30 Inizializzazione
linee  40 -  70 Ciclo che cerca "A9"
linee  80 - 100 Contatore
linee 110 - 120 Fine
```

(Poiche' Y viene incrementato prima del suo uso, il valore indice e' 1. Percio' le istruzioni che confrontano i campi degli indirizzi sono state messe a 1).

```
BFFF,Y(Y=1)=F000,Y(Y=0)
```

Fate ENTER e RUN; la funzione WATCH vi mostra i risultati: il contenuto di $0334 = contenuto di X = numero di "A9" trovati.
(Il massimo che potete trovarne e' 4, si puo' cambiare modificando la linea 90).

Utilizzo di piu' di un indice

Ora scriveremo un programma che usi entrambi i registri indice per indicizzare dati differenti contemporaneamente. Il programma crea una lista di tutti i numeri minori di $38 trovati fra gli indirizzi $F000 e $F0FF.

```
NEW
 10 A200          LDX #$00
 20 A0FF          LDY #$FF
 30 C8            INY
 40 B900F0        LDA $F000,Y
 50 C938          CMP #$38
 60 B0L90         BNE L90
 70 9D00C2        STA $C200,X
 80 E8            INX
 90 C0FF          CPY #$FF
100 D0L30         BNE L30
110 8E3403        STX $0334
120 60            RTS
WATCH             (Quale indirizzo?) 334
```

In questo caso X viene usato come puntatore (indice) alla locazione dove sono memorizzati i risultati e Y come puntatore alla locazione da cui sono stati letti i dati.
Per controllare i numeri minori di $38 abbiamo utilizzato CMP e BCS (A>=Mem, vedi capitolo 6) per evitare la

memorizzazione e l'incremento del puntatore alla zona in cui memorizziamo i risultati. Date ENTER e RUN, e con DUMP controllate che i numeri trovati siano minori di $38.

Indirizzamento indicizzato in pagina zero

Tutte le istruzioni viste prima sono del tipo indicizzato con indirizzamento assoluto, ma e' possibile avere questa forma di indirizzamento in pagina zero (vedi capitolo 2, indirizzamento indicizzato in pagina zero). Per riscrivere il programma precedente ed operare con i primi 255 byte della memoria e' necessario cambiare la linea 40 in LDA $00,Y; pero', se controllate la lista delle istruzioni del 6510 nell'appendice 1, troverete che non esiste un indirizzamento del tipo LDA pagina zero,Y ma solo LDA pagina zero. Abbiamo due scelte da poter fare, ma in pratica continueremo ad usare le istruzioni di indirizzamento indicizzato assoluto

```
BD 0000              LDA $0000,Y
```

In questo esercizio scambieremo le istruzioni che usano X con Y e inseriremo LDA pagina zero,X:

```
 10 A000             LDY #$00
 20 A2FF             LDX #$FF
 30 E8               INX
 40 B500             LDA $00,X
 70 9900C2           STA $C200,Y
 80 C8               INY
 90 E0FF             CPX #$FF
110 8C3403           STY $0334
LIST
ENTER
RUN
```

Cio' per dimostrare che bisogna stare molto attenti nella scelta dei registri perche', sebbene possano fare molte delle stesse cose, ci sono dei comandi che non possono essere fatti con lo stesso metodo di indirizzamento. Durante la stesura dei programmi e' buona regola riferirsi all'elenco delle istruzioni in appendice 1.

Capitolo 7 SOMMARIO

1. INC - aggiunge 1 al contenuto dell'indirizzo di memoria specificato.

2. DEC - sottrae 1 dal contenuto dell'indirizzo di memoria specificato.
3. Il flag di zero (non il carry) viene influenzato da queste istruzioni.
4. Sono usate soprattutto come contatori di ciclo per lasciare libero l'accumulatore per altri scopi.
5. Gli altri due registri multiuso del microprocessore, X e Y, si possono usare come contatori e come registri indice.
6. L'indirizzamento indicizzato somma il valore del registro specificato all'indirizzo assoluto (o in pagina zero) usato, per calcolare l'indirizzo finale del dato da utilizzare.
7. Molte istruzioni sono simili se si usano con A, X o Y, ma ve ne sono altre ed altri modi di indirizzamento che non sono possibili con tutti i registri. Quando scrivete un programma controllate che il formato dell'istruzione che volete scrivere sia quello corretto.

CAPITOLO 8

USO DI INFORMAZIONI MEMORIZZATE IN TABELLE

Uno degli usi principali dei registri indice e' quello di cercare in tabelle che possono essere utilizzate per varie ragioni: per memorizzare dei dati, per contenere gli indirizzi di routine varie o anche per permettere complesse conversioni di dati da una forma all'altra.

Visualizzazione di caratteri grafici

Una delle conversioni, per cui non e' possibile avere una formula, e' quella fra codice di schermo e forma del carattere visualizzato sullo stesso. Normalmente viene fatta dall'hardware del calcolatore e non bisogna preoccuparsene, ma quando siamo in modo grafico, questa parte dell'hardware viene disattivata. Nel modo normale, invece, il nostro ufficio postale interno allo schermo visualizza tramite delle "finestrelle" il carattere corrispondente al numero memorizzato in quella posizione. Cioe', noi vediamo quello che c'e' nella casella attraverso una specie di "filtro" che converte ciascun numero in una forma differente. Nel modo grafico, questo "filtro" e' stato tolto e quello che vediamo e' ciascun bit di ogni numero memorizzato nella memoria di schermo. Ad ogni bit attivato corrisponde un punto sullo schermo (pixel) e ad ogni bit spento un punto nero.

In altre parole il byte $11 visto come 00010001 viene stampato sullo schermo come 8 punti: tre neri, uno bianco, tre neri e uno bianco. Ogni carattere che viene

visualizzato sullo schermo e' composto da una griglia di 8X8 punti. Percio', dato che per ogni byte possiamo stampare otto punti, per stampare un carattere saranno necessari 8 byte, uno sotto l'altro.

Per esempio il carattere A equivale a:

Griglia di 8X8 pixel	byte binario equivalente	byte esadecimale equivalente
0 1 2 3 4 5 6 7		
0	00011000	18
1	00100100	24
2	01000010	42
3	01111110	7E
4	01000010	42
5	01000010	42
6	01000010	42
7	00000000	0

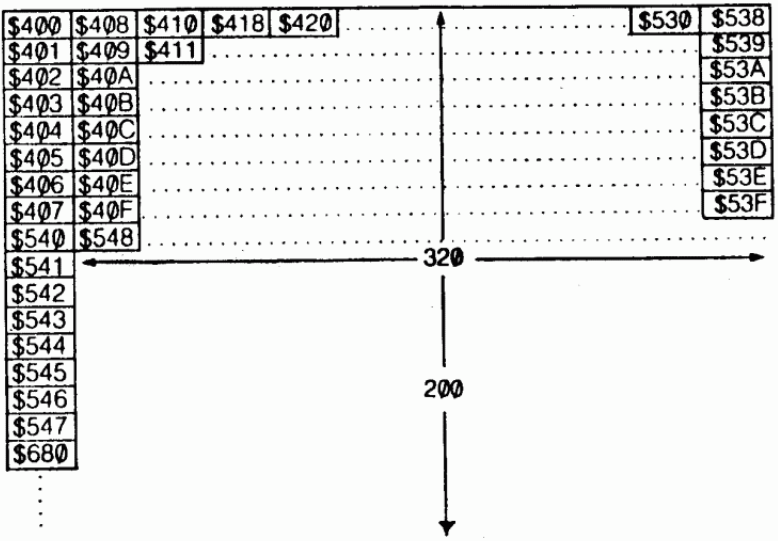
Questa stringa di otto byte: 18, 24, 42, 7E, 42, 42, 42, 0, e' esattamente quello che troviamo in quella zona di memoria detta "generatore di caratteri".

Memoria grafica

La memoria nel modo grafico non e' disposta semplicemente sullo schermo: nel modo normale abbiamo visto che $400 era l'angolo in alto a sinistra, $401 il successivo, $402 il prossimo ancora e cosi' via fino alla fine della prima riga con $427. $428 era di nuovo il primo carattere sulla seconda riga, proprio sotto $400, e cosi' via per 1000 byte (25 righe per 40 caratteri). Nel modo grafico non e' tutto cosi' semplice: il byte $400 e' sempre il piu' in alto a sinistra, ma $401 si trova sotto di lui, $402 ancora sotto, fino ad arrivare a $407. Dopo $407, $408 e' sulla riga piu' in alto di fianco a $400, $409 sara' sotto $408 e avanti cosi' fino a $40F. In questo modo la memoria di schermo viene definita per blocco-carattere (8 byte in verticale) sullo schermo. Tutto cio' viene fatto per le posizioni di tutti i 40 caratteri sullo schermo.

Cio' vuol dire che ci sono 40 byte di 8 bit (40X8=320 pixel) e che la posizione subito sotto $407 e' data da:

$400 + 40 byte orizzontali X 8 byte verticali = $400 + $320 = $400 +$140 = $540.



L'intero schermo e' formato da 320 X 200 pixel e occupa (320 X 200)/8=8000 byte di memoria. Il punto di partenza della memoria di schermo puo' essere cambiato in entrambi i modi per comodita' del programmatore (vedi appendice 6). In questo caso la memoria si sovrappone al programma BASIC che compare sullo schermo come una serie di punti (nel nostro caso si tratta dell'ALPA, ed e' di vitale importanza che non sciviamo su questo programma disegnando sullo schermo).

Abbiamo visto che il carattere A e' formato da una stringa di 8 byte e che i primi otto byte della memoria di schermo identificano la posizione di un carattere; allora, mettendo i valori sopraelencati in questi primi byte, otteniamo una A nell'angolo alto a sinistra dello schermo.

Copia del set di caratteri dalla ROM

Scrivete questo programma, che copia il set di caratteri dalla memoria caratteri a una zona in cui possano piu' facilmente essere manipolati. (Non preoccupatevi delle

istruzioni non ancora trattate).

```
NEW
 10 A9FE          LDA #$FE
 20 2D0EDC        AND $DC0E
 30 8D0EDC        STA $DC0E
 40 A9FB          LDA #$FB
 50 2501          AND $01
 60 8501          STA $01
 70 A200          LDX #$00
 80 BD00D0        LDA $D000,X
 90 9D00CF        STA $CF00,X
100 BD00D8        LDA $D800,X
110 9D00CE        STA $CE00,X
120 E8            INX
130 D0L80         BNE L80
140 A904          LDA #$04
150 0501          ORA $01
160 8501          STA $01
170 A901          LDA #$01
180 0D0EDC        ORA $DC0E
190 8D0EDC        STA $DC0E
200 60            RTS
NWATCH
ENTER
RUN
```

Abbiamo cosi' copiato i primi 32 caratteri di entrambi i set a $CE00 e $CF00 (vedi appendice 11).

Adesso scriveremo un programma che crea una lista di lettere da stampare, cerca i dati per ciascuna lettera e la stampa. Per prima cosa pero' dobbiamo scrivere un programma che definisca il messaggio da stampare.

```
NEW
 10 A908          LDA #$08
 20 8D00CD        STA $CD00
 30 A905          LDA #$05
 40 8D01CD        STA $CD01
 50 A90C          LDA #$0C
 60 8D02CD        STA $CD02
 70 8D03CD        STA $CD03
 80 A90F          LDA #$0F
 90 8D04CD        STA $CD04
100 60            RTS
ENTER
RUN
```

Ora abbiamo tre tabelle in memoria:

CD00 - Messaggio da stampare
CE00 - Set caratteri 1
CF00 - Set caratteri 2

Nel modo grafico, che useremo nel prossimo programma, vanno sottolineate alcune cose:

1. La memoria COLORE dello schermo, di cui parleremo nell'appendice 6, deve essere colorata con colori che rendano facilmente visibili i punti disegnati sullo schermo.

2. Abbiamo definito la memoria per la grafica e il colore in zone veramente inusuali, per semplicita' e per dimostrare alcune particolarita'. La memoria COLORE inizia a $400 (1024) e va fino a $800 (2048); quella di schermo va da $0 fino a $2000 (8192). Percio' sullo schermo verranno stampati dati da $0 a $2000.

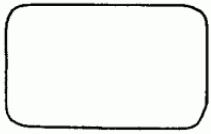
A questo punto possiamo vedere le variabili del BASIC, del sistema operativo in pagina zero (vedi appendice 9), lo stack di sistema (vedi capitolo 12), l'ALPA, il programma che abbiamo scritto in memoria come variabile BASIC e i valori nella memoria colore. Tutto e' molto poco chiaro da visualizzare, specialmente perche' ogni byte ci appare come una serie di 8 punti.

Quando vediamo apparire queste cose sullo schermo dobbiamo stare molto attenti a non cancellare o sovrascrivere questi dati. Cominceremo a scrivere nei byte a partire dall'indirizzo $400, che, come voi sapete, e' anche usata come memoria colore, e vedremo sullo schermo un carattere che cambia colore.

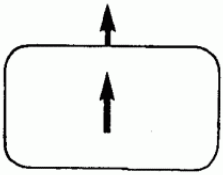
Dove appariranno i nostri disegni?

La memoria grafica, come abbiamo stabilito, inizia a $0 e noi cominciamo a disegnare a partire da $400. Poiche' ciascuna riga carattere e' (lo abbiamo gia' detto) lunga 40X8= $140 byte, i nostri disegni appariranno a $400/$140 che si trova sulla quarta riga. Disegnando 5 caratteri byte per byte, a 8 byte per carattere, abbiamo utilizzato 40 byte. Siccome la memoria colore coincide con i byte che abbiamo disegnato, i primi 40 byte di quest'ultima vengono alterati e la prima riga dello schermo apparira' con differenti colori. Accertatevi che lo schermo sia pulito,

prima di far girare il programma, per evitare che il calcolatore faccia degli scroll del video e cancelli la prima riga con i colori e le lettere stampate.



SCHERMO



La prima linea viene persa con lo scroll.

Adesso partiamo con il programma vero e proprio:

```
NEW
 1 A90F          LDA #$0F
 2 A2FF          LDX #$FF
 3 9D0004        STA $400,X
 4 CA            DEX
 5 D0L3          BNE L3
 6 A910          LDA #$10
 7 8D18D0        STA $D018
12 A93B          LDA #$3B
13 8D11D0        STA $D011
14 A000          LDY $00
15 8C00CC        STY $CC00
16 8C01CC        STY $CC01
18 BE00CD        LDX $CD00,Y
20 A9F8          LDA #$F8
22 18            CLC
24 6908          ADC #$08
26 CA            DEX
28 E0FF          CPX #$FF
30 D0L22         BNE L22
32 A8            TAY
34 AE00CC        LDX $CC00
36 A908          LDA #$08
38 8D02CC        STA $CC02
40 B900CE        LDA $CE00,Y
42 C8            INY
44 9D0004        STA $0400,X
46 E8            INX
48 CE02CC        DEC $CC02
50 D0L40         BNE L40
52 8E00CC        STX $CC00
54 AC01CC        LDY $CC01
56 C8            INY
58 C006          CPY #$06
60 D0L16         BNE L16
62 60            RTS
```

NOTA: questo programma lascia il computer nel modo grafico. Per ripristinare le condizioni iniziali premete RUN/STOP e RESTORE contemporaneamente, poi fate GOTO 2020.

Riepilogo del programma

Linee 1 - 5	attivano i colori
Linee 12 - 13	attivano il modo grafico
Linee 14 - 16	mettono a zero il puntatore alla posizione di stampa (a CC00) e leggono i caratteri per il messaggio (CC01).
Linee 16 - 18	prendono il prossimo carattere da stampare
Linee 20 - 30	trovano l'inizio dei dati per il carattere da stampare moltiplicando il valore per 8. (8 byte per carattere)
Linea 32	TAY. Questa istruzione trasferisce il contenuto dell'accumulatore nel registro Y. (Transfer accumulator to Y)(ne parleremo piu' avanti nel capitolo)
Linee 36 - 38	attivano un contatore per limitare la copia dei dati a SCC02.
Linee 40 - 50	copiano i dati dal loro posto in tabella al loro posto sullo schermo, indicizzando entrambi i puntatori
Linea 52	memorizza il puntatore alla posizione di stampa sullo schermo per usarlo per il prossimo carattere ($CC00).
Linee 54 - 60	incrementano il puntatore alla lista di lettere che costituiscono il messaggio e ciclano a stampare il prossimo, finche' non sono stati stampati tutti
Linea 62	fine.

Fate ENTER, premete CLR e poi date RUN.

Vedrete stampato un messaggio "HELLO" vicino all'angolo sinistro in alto dello schermo.

Indirizzamento indicizzato indiretto

Ci sono dei casi in cui non sarete sicuri circa la scelta della tabella dove trovare i dati. In altre parole, immaginate un programma che vi lasci la scelta, in qualsiasi momento, di stampare il messaggio con il set di caratteri maiuscolo o minuscolo. Volete decidere quale

tabella usare a meta' programma: cio' potrebbe essere fatto con due programmi identici che accedono a differenti tabelle e con all'inizio del programma la scelta di quale dei 2 sottoprogrammi usare. Naturalmente questo metodo e' molto dispendioso per quanto riguarda la memoria; per accedere ai dati in questa maniera c'e' un metodo di indirizzamento, detto Indicizzato indiretto, che permette di avere una maggiore flessibilita' circa la posizione in cui mettere i dati. L'indirizzamento indicizzato indiretto e' come l'indicizzato assoluto, eccetto che l'indirizzo non e' parte dell'istruzione, ma e' contenuto in due locazioni successive in pagina zero. In altre parole, le due locazioni in pagina zero sono l'indirizzo di una cella di memoria che viene indicizzata con il registro Y per ottenere l'indirizzo finale dell'istruzione. (Il metodo di indirizzamento indicizzato indiretto usa sempre il registro indice Y).

Per spiegarlo con l'esempio dell'ufficio postale fate conto di avere un'istruzione che guarda in una casella (pagina zero); il numero trovato in questa e nella successiva viene sommato per trovare l'indirizzo assoluto (nella forma byte basso - byte alto). A questo indirizzo, cosi' trovato, sommate un'indice (Y) per trovare la casella in cui dovete effettivamente guardare.

Il codice mnemonico di questa istruzione e': QQQ(PZ),Y
dove QQQ e' un'istruzione, ad esempio LDA,
PZ e' l'indirizzo di un byte in pagina zero
e la Y fra parentesi indica che prima si trova l'indirizzo e poi si aggiunge l'indice.

Il programma maiuscolo/minuscolo memorizza solo l'indirizzo di partenza della tabella corretta (indirizzandolo in pagina zero) e chiama la routine generale.

Modifichiamo il programma precedente aggiungendo queste linee:

```
40 B1FD          LDA ($FD),Y
 8 A9CF          LDA #$CF
 9 85FE          STA $FE
10 A900          LDA #$00
11 85FD          STA $FD
LIST
ENTER
RUN
```

Premete RUN/STOP e RESTORE, per uscire, e fate GOTO 2020.

Adesso, cambiando la linea 8 in: 8 A9CE LDA #$CE, questo stesso programma cambia il set di caratteri. Scrivete la nuova linea 8 poi fate ENTER e RUN.

Giusto per informazione, provate a cambiare la linea 8 dando un valore che non sia ne' $CE ne' $CF, ad esempio $47. Cio' fa si' che il programma vada a cercare i dati ad indirizzi che non sono quelli in cui abbiamo messo i dati. NOTA: il programma lavora lo stesso, ma stampa delle cose senza senso; il calcolatore non conosce la differenza fra un dato senza senso e uno corretto.

Istruzioni sul trasferimento di registri

Indietro, alla linea 32 del nostro programma, abbiamo incontrato un'istruzione che non avevamo ancora visto, TAY (Transfer A into Y - Trasferisci A in Y): essa non e' che una delle semplici istruzioni che servono per copiare il contenuto di un registro in un altro.

Le istruzioni possibili sono:

```
TAX (Transfer A into X - Trasferisci A in X)
TAX (   "     A  "   Y -     "       A  " Y)
TXA (   "     X  "   A -     "       X  " A)
TYA (   "     Y  "   A -     "       Y  " A)
```

Queste istruzioni sono usate principalmente quando le operazioni da fare su un contatore o un indice richiedono manipolazioni matematiche che debbono essere fatte sull'accumulatore per poi riportare il risultato nel registro indice.

NOTA: non esistono istruzioni di trasferimento fra X ed Y; se necessario bisogna passare prima da A.

Esistono altri due metodi di indirizzamento che non abbiamo ancora trattato e di cui daremo una breve spiegazione adesso: il primo e' detto Indirizzamento indicizzato indiretto (no, non e' quello che abbiamo appena trattato, e cioe' l'indirizzamento indiretto indicizzato, che si scrive nella forma QQQ(PZ),Y dove l'indirizzamento viene fatto prima dell'indicizzazione); nell'indicizzato indiretto, QQQ(PZ,X), l'indicizzazione viene fatta prima di calcolare l'indirizzo in pagina zero che contiene il primo dei due byte dell'indirizzo (byte basso - byte alto) dell'eventuale destinazione dell'istruzione.

Immaginate di avere una tabella di indirizzi in pagina zero, che puntano a dati o tabelle differenti. Per trovare il primo byte di queste tabelle prendete queste istruzioni per indicizzare attraverso la tabella in pagina zero e usate il corretto indirizzo per trovare la tabella o il dato che vi interessa. Per usare ancora la similitudine dell'ufficio postale prendete il numero di una casella postale (pagina zero); a questo indirizzo sommate il valore del registro indice (Registro X) e trovate l'indirizzo il cui contenuto, sommato al contenuto della casella successiva (byte basso - byte alto), da' l'indirizzo dove si trovano effettivamente i dati cercati.

Indirizzamento indiretto

E' l'ultimo tipo di indirizzamento possibile e viene usato solo con l'istruzione JMP.

Il comando JMP, usato con l'indirizzamento assoluto, fa saltare il programma all'indirizzo specificato nell'istruzione (come il GOTO del BASIC).

Con l'indirizzamento indiretto, JMP (Indirizzo), l'indirizzo da due byte (assoluto) all'interno delle parentesi viene usato per puntare ad un indirizzo in memoria che e' il byte basso di un indirizzo da due byte dove deve saltare il programma. In altre parole, l'istruzione punta ad un indirizzo in memoria che, con il successivo, da' l'indirizzo del salto. In termini di paragone con l'ufficio postale significa: guardate all'interno di una casella e nella successiva, mettete insieme i contenuti trovati e otterrete l'indirizzo a cui deve saltare l'istruzione. L'uso maggiore di questa istruzione e' quello conosciuto come input o output vettorizzato. Per esempio, se scrivete un programma che salta direttamente all'indirizzo in ROM dell'output caratteri, per stampare un carattere e volere che l'uscita venga inviata direttamente al disco, bisogna cambiare l'istruzione JMP. Usando l'output vettorizzato il programma fa un salto ad una locazione RAM. Se il sistema operativo del disco e' chiamato a prendere il controllo dell'output, detto sistema prepara il vettore delle locazioni, potendo fare cosi' un salto indiretto ai suoi programmi. Se l'output e' diretto allo schermo, dette locazioni di memoria conterranno gli indirizzi delle routine di stampa della ROM, e il programma uscira' attraverso esse. Adesso possiamo fare un elenco di tutti i metodi di indirizzamento

possibili con il 6510:

Implicito		QQQ
Assoluto		QQQ indirizzo
Pagina zero		QQQ PZ
Immediato		QQQ # byte
Relativo		BQQ byte - (L# per ALPA)
	Assoluto,X	QQQ indirizzo,X
	Assoluto,Y	QQQ indirizzo,Y
Indicizzato	Pagina zero,X	QQQ PZ,X
	Pagina zero,Y	QQQ PZ,Y
Indiretto indicizzato		QQQ (PZ),Y
Indicizzato indiretto		QQQ (PZ,X)
Indiretto		JMP (indirizzo)
e anche		
Accumulatore		QQQ A

(Operazione possibile sull'accumulatore, vedi capitolo 10).

Capitolo 8 SOMMARIO

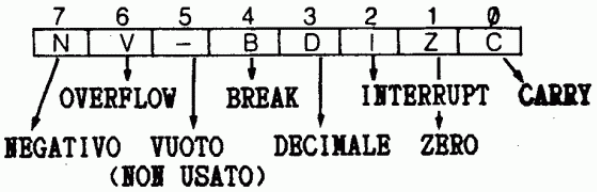
1. Nel modo grafico si puo' vedere il contenuto della memoria di schermo. Un bit equivale a un pixel (un punto sullo schermo).
2. I caratteri sono definiti in blocchi da 8X8 pixel.
3. La memoria di schermo in modo grafico opera sullo schermo in blocchi di caratteri, linea per linea.
4. Il set di caratteri e' messo in ROM al di sopra della memoria, dall'indirizzo $D000 in poi.
5. I registri indice vengono usati, fra l'altro, per cercare su tabelle usando piu' metodi di indirizzamento indicizzato.
6. Nel normale indirizzamento indicizzato il registro indice e' sommato ad un indirizzo assoluto (o in pagina zero) per calcolare l'eventuale indirizzo.
7. Nell'indirizzamento indiretto indicizzato l'eventuale indirizzo e' calcolato sommando il registro Y all'indirizzo da due byte memorizzato in una locazione in pagina zero, puntato dall'indirizzo da un byte dell'istruzione.
8. Nell'indirizzamento indicizzato indiretto, l'eventuale indirizzo viene ricavato sommando il registro indice X all'indirizzo in pagina zero, che e' parte dell'istruzione. Il contenuto di queste due locazioni in pagina zero specifica l'indirizzo.
9. Il calcolatore non puo' dirci la differenza fra dati

senza senso o dati validi.
10. TAX, TAY, TXA e TYA sono usate per trasferire i dati fra i registri e l'accumulatore.
11. L'indirizzamento indiretto (solo per JMP) utilizza il contenuto di due byte (uno successivo all'altro), ovunque siano in memoria, come indirizzo della destinazione del salto.

CAPITOLO 9

CODICI DELLO STATO DEI PROCESSI

Abbiamo menzionato, nei capitoli 5 e 6, il concetto di flag del microprocessore e,parlando del flag di carry, del flag di zero, delle istruzioni "branch" e di altre istruzioni associate a questi flag, ad esempio SEC, CLC, BCS, BCC, BEQ e BNE. Abbiamo detto che questi si trovano, insieme ad altri, in un particolare registro chiamato Registro dei codici dello stato dei processi o piu' semplicemente Registro di stato. Esso e', come gli altri registri o byte in memoria, composto da 8 bit (dita); ciascun bit rappresenta un flag con scopi diversi:



Una lista delle istruzioni che influenzano i flag la potete trovare nell'appendice 1.

1. Il flag di carry (C) - viene attivato o meno, come abbiamo gia' detto, per indicare un riporto o un prestito dell'ottavo bit di un byte, funzionando come un nono bit per intervenire in calcoli futuri o per essere ignorato. Il carry si puo' attivare o disattivare con SEC e CLC, rispettivamente, e un programma puo' controllare il suo stato con BCS o BCC.

2. Il flag di zero (Z) - viene attivato o disattivato in dipendenza del risultato di un'operazione, del confronto di numeri o del trasferimento di dati; il suo stato viene controllato con BEQ o BNE.

3. Il flag di break (B) - il comando BRK provoca quella che viene chiamata un'interrupt (interruzione); maggiori dettagli su di esso si trovano nel capitolo 11. Con il comando BRK, il linguaggio macchina si blocca e il calcolatore compie un salto indiretto ai contenuti delle locazioni $FFFE e $FFFF. Questi indirizzi della ROM

contengono l'indirizzo di una routine di interruzione che rimanda il controllo al BASIC. Il comando BRK e' uno dei piu' usati per la messa a punto di programmi: collocando questo comando in qualsiasi punto del programma in L.M. potete controllare come viene fatta una data operazione o perche' venga sbagliata, oppure interrompere il programma prima che faccia operazioni che provocano errori. Provate ad usare il comando nei listati proposti dal capitolo 7.

4. Il flag di interrupt (I) - si puo' modificare con SEI e CLI; quando e' attivato disabilita alcuni tipi di Interrupt (interruzioni) che possono capitare (vedi capitolo 11).

5. Il flag di decimale (D) - viene attivato o no con SED e CLD rispettivamente. Quando il flag e'attivato,il computer passa al modo BCD, dove BCD sta per Binary coded decimal che e' un metodo per rappresentare i numeri decimali in memoria. Con la rappresentazione BCD, le cifre esadecimali da 1 a 9 sono lette come la loro equivalente decimale e le cifre da A a F non hanno significato. In altre parole:

RAPPRESENTAZIONE BCD

BINARIO	ESA	VALORE DECIMALE BCD
00000000	00	0
00000001	01	1
00000010	02	2
00000011	03	3
00000100	04	4
00000101	05	5
00000110	06	6
00000111	07	7
00001000	08	8
00001001	09	9
00010000	10	10
00010001	11	11
00100010	22	22
01000011	43	43
10011000	98	98

Lo schema fa vedere come ci siano 6 codici fra 9 e 10 che vengono persi.

Nel modo decimale il calcolatore somma e sottrae

automaticamente numeri BCD:

Flag decimale = 0	Flag decimale = 1
17	17
+26	+26
3D	43

Il problema, con il modo decimale, e' che si spreca memoria e che e' molto lento da usare in matematica (se si escludono addizioni e sottrazioni). Nel complesso e' piu' semplice usare l'esadecimale e convertirne i risultati. Provate a modificare alcuni programmi del capitolo 5 per farli lavorare in decimale e confrontate i risultati con quelli ottenuti normalmente.

6. Il flag di negativo (N). - Finora abbiamo sempre detto che i numeri che possono essere memorizzati in un byte devono essere compresi fra 0 e 255, e che quelli maggiori di 255 devono usare due byte; pero' non abbiamo mai parlato di numeri negativi, ma li abbiamo visti solo di sfuggita nel capitolo 6. I numeri fra 0 e 255 sono usati dal calcolatore alla stessa maniera sia che rappresentino indirizzi, caratteri o numeri BCD. Nel capitolo 6 abbiamo preso il nostro numero compreso fra 0 e 255 e abbiamo deciso di usarlo come valore di un salto condizionato (branch); abbiamo anche stabilito che un salto in avanti (positivo) veniva identificato con un numero compreso fra $00 e $7F e un salto indietro (negativo) con un numero fra $80 a $FF. Questo sistema di numerazione e' puramente arbitrario, ma, dato che funziona, e' un'operazione matematica possibile per rappresentare numeri positivi e negativi. Il sistema che viene usato si chiama Complemento Due o Secondo Complemento. La tabella dell'appendice 4 serve per le conversioni fra numeri normali e numeri del secondo complemento; cercate al centro il numero decimale da convertire e troverete ai bordi della tabella il corrispondente numero esadecimale del secondo complemento. In termini matematici si tratta del complemento del numero binario (tutti gli 1 diventano 0 e viceversa) a cui si aggiunge 1. Esempio:

COMPLEMENTO

3 = 0 0 0 0 0 0 1 1 → [1|1|1|1|1|1|0|0]
+1
= [1|1|1|1|1|1|0|1] = FD = −3

Con questa rappresentazione, ogni byte che ha un valore maggiore di 127 (con il settimo bit attivato), rappresenta un numero negativo e un byte con un valore minore di 128 (il bit piu' alto disattivato), rappresenta un numero positivo.

```
1 X X X X X X X  -  NEGATIVO
0 X X X X X X X  -  POSITIVO
```

Il flag di negativo e' un registro di stato che si attiva automaticamente (come il flag di zero) se il risultato di un'operazione, il confronto o il trasferimento sono negativi. Percio' il microprocessore non puo' dirci se un valore rappresenta un numero, un carattere o qualcos'altro ancora; egli pero' attivera' sempre il flag di negativo se il settimo bit del byte in questione e' attivato. In altre parole il flag di negativo e' una copia del settimo bit del risultato di un'operazione.

Quindi il bit piu' alto (settimo) di un byte e' il bit del segno (rappresenta il segno del numero) e rimangono disponibili 7 bit per rappresentare il numero in questione. Con questi 7 bit si puo' rappresentare un numero compreso fra 0 e 127, (incluso 0 = -0) aggiungendo un uno nel negativo. I numeri in complemento a due possono rappresentare numeri compresi fra -128 e +127 usando un byte.

Provate a fare queste operazioni matematiche usando il nuovo sistema:

Complemento a due Binario	Valore decimale	
Positivo + Positivo (tutto normale)		
00000111	+ 7	
+00001001	++ 9	
00010000	16	C = 0 V = 0 N = 0
Positivo + Negativo (risultato negativo)		
00000111	+ 7	
+11110100	+-12	
11111011	- 5	C = 0 V = 0 N = 1
Positivo + Negativo (risultato positivo)		
00000111	+ 7	
+11111101	+- 3	
(1)00000100	+ 4	C = 1 V = 0 N = 0

Positivo + Positivo (risposta maggiore di 127)

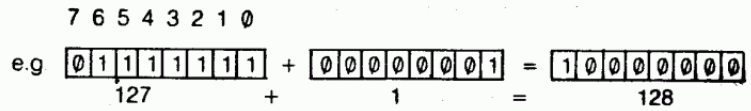
```
  01110011                  115
 +00110001                + 49
 ---------                ----
  10100100                 -92  C = 0 V = 1 N = 1
 ---------                ----
```

NOTA: la risposta e' sbagliata!

La numerazione in complemento a due sembra voler funzionare perfettamente ad eccezione dell'ultimo esempio. Come abbiamo detto, in complemento a due si possono rappresentare numeri fra -127 e +127 e, nel caso dell'esempio sbagliato, il risultato e' 164; come succedeva nel capitolo 3 per i numeri maggiori di 255, anche qui abbiamo bisogno di 2 byte. Nel normale binario il riporto del bit 7 (il piu' alto) nel byte alto, veniva fatto tramite il carry; in complemento a due abbiamo sempre 7 bit, ma il settimo rappresenta il segno, quindi il bit piu' alto diventa il 6. Il microprocessore non sa che noi usiamo il complemento a due ed ha, come al solito, riportato il bit 6 nel bit 7. Per svolgere questa operazione correttamente, cioe' tenendo conto del fatto che si usa il complemento a due, il microprocessore ha un flag di overflow (eccesso) per segnalarci quello che e' appena successo.

7. Il flag di overflow (V) - Esso viene attivato quando c'e' un riporto del bit 6 nel bit 7.

```
        7 6 5 4 3 2 1 0
e.g  [0|1|1|1|1|1|1|1] + [0|0|0|0|0|0|0|1] = [1|0|0|0|0|0|0|0]
        127          +        1          =        128
```



L'uso maggiore di questo flag viene fatto per segnalare il cambio accidentale del segno, causato da un "overflow", usando l'aritmetica in complemento a due. Per correggere questo cambio al bit del segno (bit 7) viene sostituito il suo complemento (praticamente viene invertito) e viene fatto un riporto sul bit piu' alto, se necessario.

Cio' fa si' che la risposta sbagliata di -92 (10100100) diventi 1X128 (byte alto) + 36 (00100100). 128+36=164 che e' il risultato giusto.

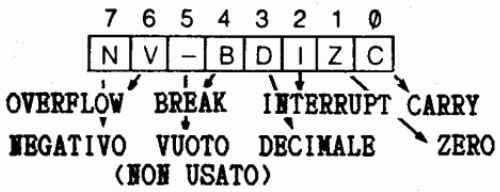
Un programma puo' controllare se il flag di negativo e' attivato o no con BMI (Branch on minus - Salta se negativo) e BPL (Branch on plus - Salta se positivo) rispettivamente.

Il test per controllare se il flag di overflow e' attivo o no viene fatto con BVS (Branch on overflow set - Salta se overflow e' attivo) o BVC (Branch on overflow clear - Salta

se overflow non e' attivo) rispettivamente. Il flag di overflow puo' essere disattivato con CLV (Clear overflow - Disattiva overflow).

Capitolo 9 SOMMARIO

1. Il microprocessore contiene un registro speciale detto Registro dei codici dello stato dei processi.



2. CARRY - SEC, CLC
BCS, BCC
Istruzioni attive se ci sono le condizioni necessarie.
3. ZERO - BEQ, BNE
Attive se il risultato di un trasferimento e' 0.
4. BRK - E' un'istruzione che attiva il flag di break e ferma il microprocessore (molto utile per la messa a punto di programmi).
5. INTERRUPT - SEI, CLI
Vedi capitoli 11 e 12.
6. DECIMALE - SED, CLD
Attivano il modo decimale; le addizioni e le sottrazioni sono fatte con il modo BCD.
7. La numerazione in complemento a due rappresenta numeri fra -128 e +127.
negativo X = (complemento (X)) +1
8. NEGATIVO - E' attivato se il bit 7 del risultato e' attivo (=1)
BMI, BPL
9. OVERFLOW - E' attivato dal riporto (carry) del complemento a due.
CLV
BVS, BVC

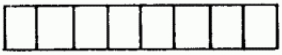
CAPITOLO 10

OPERATORI LOGICI E MANIPOLATORI DI BIT

Cambiamento di bit in memoria

In questo capitolo tratteremo di un gruppo di istruzioni molto diverse da quelle discusse finora, ma allo stesso tempo fondamentali per il lavoro del calcolatore: le operazioni "logiche" o "booleane". Sono i comandi: AND (AND logico), ORA (OR logico) e EOR (OR esclusivo logico). Queste operazioni sono fatte con semplici circuiti e la maggior parte delle funzioni del computer sono svolte con serie di questi circuiti. Dette operazioni logiche dei circuiti sono accessibili tramite le citate istruzioni, ed e' di questo che ci occuperemo, non dell'hardware, in questo capitolo.

Noi sappiamo che i byte e i registri della memoria sono composti da 8 bit:



Per spiegare queste istruzioni osserviamo il loro funzionamento su un solo bit e supponiamo che la stessa operazione venga fatta su tutti gli altri contemporaneamente. Un operatore logico e' come una funzione matematica che prende due dati e ne ricava un unico risultato. Esempio:

4 + 5 = 9

In questo caso, il dato in arrivo e' formato da un valore da un bit (o e' 1 o e' Ø). Per definire delle funzioni logiche abbiamo disegnato una "tavola della verita'" che mostra tutti i possibili arrivi e tutte le combinazioni in uscita.

INPUT 1 / INPUT 2	Ø	1
Ø	OUTPUT FOR Ø, Ø	OUTPUT FOR Ø, 1
1	OUTPUT FOR 1, Ø	OUTPUT FOR 1, 1

AND logico

La prima istruzione che consideriamo e' AND che effettua un AND logico fra l'accumulatore e la memoria lasciando il risultato in A. Il risultato di un AND logico e' 1 se la prima cifra e (in inglese AND, da cui il nome) la seconda in entrata sono 1. La "tabella della verita'" e':

ACCUMULATOR \ MEMORY	Ø	1
Ø	Ø	Ø
1	Ø	1

che estesa ad un byte di otto bit significa:

	Ø	1	1	Ø	1	Ø	1	1
AND	1	Ø	1	1	1	Ø	1	Ø
=	Ø	Ø	1	Ø	1	Ø	1	Ø

Se il risultato e' uguale a Ø viene attivato il flag di zero. Per esempio: se non ci sono "1" coincidenti nei due byte in questione.

L'istruzione AND e' molto utile per creare delle "maschere" (MASK) che disattivino certi bit all'interno di un byte. Supponiamo che si voglia disattivare (mettere a zero), entro un byte qualsiasi, il terzo, il quinto e il sesto bit. Dobbiamo creare una maschera che abbia attivi tutti i bit meno quelli sopracitati e fare un AND logico con il byte in questione.

	7	6	5	4	3	2	1	Ø	
Mask =	1	Ø	Ø	1	Ø	1	1	1	= $97

AND #$97

Cio' disattiva il terzo, il quinto e il sesto bit di qualsiasi byte che si trovi nell'accumulatore.

OR logico

La seconda istruzione e' ORA, che svolge un OR logico dell'accumulatore con la memoria, lasciando il risultato nell'accumulatore.

La funzione OR da' in uscita un 1 se un'entrata oppure

(in inglese OR, da cui il nome) l'altra sono uguali a 1. La tabella e':

OR

ACCUMULATOR \ MEMORY	0	1
0	0	1
1	1	1

Che estesa ad un byte da 8 bit significa:

	0	1	0	1	0	0	1	0
ORA	0	0	1	1	1	0	1	0
=	0	1	1	1	1	0	1	0

Il flag di zero viene attivato se entrambi i byte usati hanno come risultato zero.

L'istruzione ORA viene usata come la precedente per fare operazioni di mascheramento all'interno di byte.

Supponiamo di voler disattivare il secondo, il terzo e il settimo bit di un byte: dobbiamo usare una maschera con attivi solo i detti bit.

	7	6	5	4	3	2	1	0		
Mask =	1	0	0	0	1	1	0	0	=	$8C

ORA #$8C

Questo attiva il secondo, il terzo e il settimo bit di qualsivoglia cosa nell'accumulatore.

OR esclusivo logico

L'ultimo operatore logico e' EOR che fa un OR esclusivo logico della memoria con l'accumulatore, lasciando il risultato in A. La funzione OR esclusiva da' in uscita 1 se la prima entrata e' 1 o se lo e' la seconda, ma da' 0 se sono entrambe 1. La tabella e':

EOR

ACCUMULATOR \ MEMORY	0	1
0	0	1
1	1	0

Che esteso ad un byte di 8 bit significa:

	1	0	1	1	1	0	0	1
EOR	1	0	1	0	0	1	0	1
=	0	0	0	1	1	1	0	0

L'OR esclusivo si usa per fare con delle maschere il complemento (inverso) di alcuni bit all'interno di un byte.

Per invertire il primo, il secondo e il quarto bit, si deve usare una maschera con gli stessi bit attivi.

	7	6	5	4	3	2	1	0	
Mask =	0	0	0	1	0	1	1	0	= $16

EOR #$16

Cio' inverte gli stessi bit nell'accumulatore.

Scrivete questo programma con ALPA, allo scopo di verificare il funzionamento delle istruzioni appena trattate:

```
NEW
10 A9CA              LDA #$CA
20 299F              AND #$9F
30 8D3403            STA #$0334
40 A9A2              LDA #$A2
50 0984              ORA #$84
60 4D3403            EOR $0334
70 8D3403            STA $0334
80 60                RTS
WATCH   (indirizzo? 334)
```

Riepilogo del programma

```
Linea 10              A= $CA     11001010
Linea 20  AND $9F     A= $8A     10001010
Linea 30    mem.      A= $334    10001010
Linea 40              A= $A2     10100010
Linea 50  ORA $84     A= #A6    10100110
linea 60  EOR $334    A= $2C    00101100
```

Fate ENTER e RUN, e verificate se i risultati corrispondono con quelli dati.

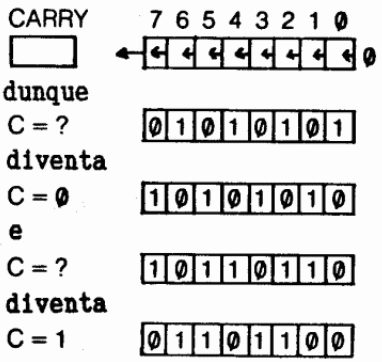
L'istruzione BIT

E' un'istruzione che effettua dei controlli o dei confronti

in un modo molto interessante. Nel capitolo 6 abbiamo incontrato il comando CMP che faceva una sottrazione modificando i flag di stato senza memorizzare il risultato. In maniera simile BIT (confronta bit in memoria con l'accumulatore), facendo un AND logico fra l'accumulatore e la memoria, modifica come risultato il flag di zero e mette il bit 7 nel flag di negativo e il bit 6 in quello di overflow.

Rotazione di bit in un byte

Adesso parleremo di altri 4 manipolatori di bit e delle conseguenze di queste manipolazioni. La prima istruzione che troviamo e' ASL (Aritmetic shift left - Spostamento a sinistra aritmetico), che sposta a sinistra di un bit tutti i bit, mettendo uno 0 nel bit piu' basso e spostando il bit piu' alto nel flag di carry.



Nel capitolo 3, quando spiegavamo l'esadecimale e il binario, abbiamo detto che ciascun bit ha il valore di 2 elevato alla posizione-1.

i.e.	128	64	32	16	8	4	2	1

Come avrete notato il valore di ogni cella e' il doppio del valore della cella precedente; percio':

00000001 X 2 = 00000010 e
00001000 X 2 = 00010000

e ancora

00111001 X 2 = 01110010

L'operazione necessaria per moltiplicare un byte per due e' quella fatta da ASL.
Per usare l'esempio fatto prima:

```
C=?  01010101 ($55)X2  da'  C=0  10101010 ($AA)
     e
C=?  10110110 ($B6)X2  da'  C=1  01101100 ($6C + CARRY)
```

Scrivete questo programma:

```
NEW
10 A90A              LDA #$0A
20 0A                ASL A
30 8D3403            STA $0334
40 60                RTS
WATCH      (indirizzo? 334)
ENTER
RUN
```

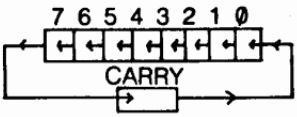
La linea 20 usa il metodo di indirizzamento dell'accumulatore, prendendo il contenuto dello stesso come dato e riportandovi il risultato.
NOTA: e' differente dall'indirizzamento implicito, perche' ASL puo' essere usata anche con la memoria.
Questa istruzione si usa per fare moltiplicazioni per potenze di due (1, 2, 4, 8,...). Per sostituire al programma precedente la moltiplicazione per 8 con quella per 2, battete le seguenti linee:

```
15 0A                ASL A
25 0A                ASL A
ENTER
RUN

$0AX8=50
```

Rotazione con il carry

Come succedeva per le addizioni, anche qui per moltiplicare numeri maggiori di 255 (numeri da due o piu' byte) occorre trovare un comando che usa il carry sia all'entrata sia all'uscita dello spostamento:



L'istruzione che fa cio' e' ROL (Rotate one bit left - Ruota un bit a sinistra). Per moltiplicare per 4 un numero di due byte fate:

```
 5 A917          LDA #$17
 6 8D3503        STA $0335
18 2E3503        ROL $0335
25 2E3503        ROL $3503
LIST
```

Aggiungendo queste linee, moltiplichiamo il numero $170A per 4.

NOTA: 1. Per evitare il cambiamento improvviso dei registri abbiamo usato ROL con l'indirizzamento assoluto, che memorizza il risultato in memoria.

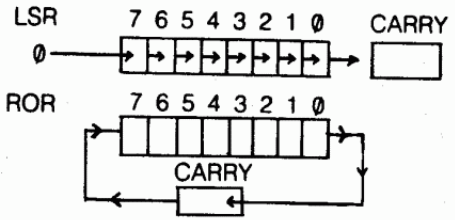
2. Abbiamo ruotato entrambi i byte una volta e poi ruotato ancora. Ruotare il byte basso due volte di seguito e poi fare la stessa cosa con il byte alto non va bene, perche' il bit alto del byte basso viene perso quando il carry e' usato per la seconda ASL.

Fate ENTER e RUN e controllate con
WATCH (indirizzo? 335) e poi ridate
RUN.

Mettete insieme il byte basso con quello alto e controllate se il risultato e' uguale al numero originale moltiplicato per quattro.

Rotazione a destra

LSR e ROR sono le istruzioni equivalenti a ASL e ROL, eccetto che loro spostano i bit nella direzione opposta.



Dato che le loro opposte possono essere pensate come moltiplicazioni per due, queste saranno divisioni per due e possono estendersi all'aritmetica multibyte. Dopo la

divisione il numero rimasto nel byte rappresenta la parte intera del risultato e il bit che e' uscito a causa dello spostamento rappresenta il resto. Esempio:

```
      $1D ÷ $08   =  3        resto 5

      00011101    = 29        resto
LSR ÷ 2
      00001110    = 14      → 1=1
LSR ÷ 4
      00000111    =  7      →01 = 1
LSR ÷ 8
      00000011    =  3      →101 = 5
```

Nota: le istruzioni di spostamento e rotazione si usano per l'aritmetica, ma non bisogna dimenticare il loro uso per la trasposizione di bit, ad esempio lo spostamento di bit nel carry per effettuare test di controllo.

Moltiplicazione intelligente

Abbiamo appena detto che spostando bit, si puo' moltiplicare per ogni potenza di 2 (1, 2, 4, 8,...128), che sono poi gli stessi valori che rappresentano ciascun bit all'interno del byte. Come sappiamo dal capitolo 3, sommando questi valori si puo' produrre qualsiasi numero compreso fra 0 e 255.

Se operiamo una moltiplicazione per ciascuno di questi valori e sommiamo il risultato, e' come se moltiplicassimo per ogni valore fra 0 e 255. Esempio:

```
$16 X $59 = 00010110 X $59
          = 00010000 X $59
          + 00000100 X $59
          + 00000010 X $59
= 16 X $59 + 4 X $59 + 2 X $59
```

Questo e' l'algoritmo che useremo per fare delle routine di moltiplicazione generalizzate. Ruoteremo (moltiplicare per due) ogni numero e lo sommeremo al totale, per ogni bit attivo dell'altro byte.

```
10110 x $59
rotate  $59                      1 0 1 1 [0]
rotate  $59  sommo al totale     1 0 1 [1] 0
rotate  $59  sommo al totale     1 0 [1] 1 0
rotate  $59                      1 [0] 1 1 0
rotate  $59  sommo al totale     [1] 0 1 1 0
```

Per semplicita' limitiamo la nostra routine di moltiplicazione generalizzata a risultati minori di 255.
Per moltiplicare $1B per $09 fate:

```
NEW
 10 A91B          LDA #$1B
 20 8D3403        STA $0334
 30 A909          LDA #$09
 40 8D3503        STA $0335
 50 A900          LDA #$00
 60 6E3503        ROR $0335
 70 2E3503        ROL $0335
 80 4E3403        LSR $0334
 90 90L120        BCC L120
100 18            CLC
110 6D3503        ADC $0335
120 D0L70         BNE L70
130 8D3603        STA $0336
140 60            RTS
```

Riepilogo del programma

Linee 10 - 60	inizializzano i valori da moltiplicare e mettono a zero il totale. ROR seguito da ROL al primo passaggio non ha alcun effetto, ma solo ROL e' all'interno del ciclo.
Linea 70	ad eccezione della prima volta moltiplica uno dei numeri (2) per due.
Linee 80 - 90	ruota l'altro numero (1) bit per bit nel carry e lo controlla per vedere se l'altro numero (2) deve essere sommato. Se il carry e' a zero viene controllata la possibilita' che il numero (1) sia stato trasposto completamente (=0 la moltiplicazione e' finita) andando alla linea 120.
Linee 100 - 110	sommano al totale (in A) il numero (2) che e' stato moltiplicato per due nel ciclo.
Linea 120	se e' stato fatto il salto da linea 90, controlla la fine della moltiplicazione. Se il salto da linea 90 non e' stato fatto la condizione dell'istruzione e' vera e quindi stiamo sommando un numero diverso da zero ad un totale minore di 255.
Linee 130 - 140	FINE.

NOTA: questa routine di moltiplicazione e' molto piu' efficiente di quella data nel capitolo 7, che ripeteva il ciclo almeno nove volte; questa usa 9 come numero (1) e $1B come numero (2) e il ciclo viene ripetuto per quattro volte (numero necessario che rappresenta 9 = 1001).

Fate:

```
WATCH   (indirizzo? 336)
ENTER
RUN
```

e verificate il risultato.

Adesso cambiate i numeri per provare calcoli diversi (ricordate che la risposta deve essere minore di 255

Esempio:

```
10 A906            LDA #$06
30 A925            LDA #$25
ENTER
RUN
```

e verificate i risultati.

Capitolo 10 SOMMARIO

1. AND

	0	1
0	0	0
1	0	1

molto spesso usata per mascherare bit disattivati.

2. ORA

	0	1
0	0	1
1	1	1

usata per mascherare bit attivi.

3. EOR (OR esclusivo)

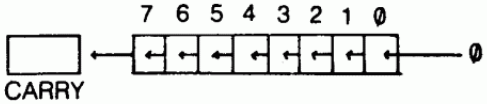
	0	1
0	0	1
1	1	0

usata per invertire i bit.

4. BIT fa un AND senza memorizzare il risultato.

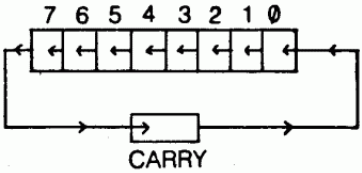
Z viene attivato o disattivato
N diventa uguale al bit 7
V diventa uguale al bit 6

5. ASL Spostamento aritmetico a sinistra

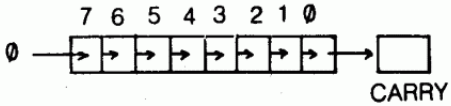


usata per moltiplicare per 2.

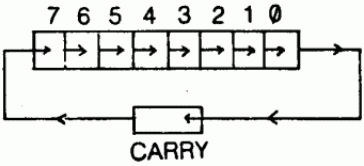
6. ROL Rotazione a sinistra di un bit



7. LSR Spostamento logico a destra



8. ROR Rotazione a destra di un bit



CAPITOLO 11

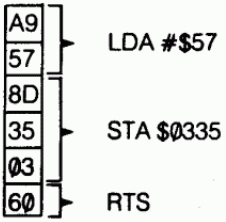
DETTAGLI SUL CONTROLLO DEI PROGRAMMI

Il program counter

Abbiamo parlato molto delle diverse operazioni che il microprocessore puo' eseguire, ma non abbiamo detto nulla su come svolge queste operazioni. Cio' e' perfettamente logico perche' in molti casi non ci interessa saperlo; ad ogni modo il conoscere come il microprocessore opera ci offre un nuovo elenco di comandi e un'area molto importante delle sue capacita'.

Il microprocessore contiene un registro di due byte che svolge funzioni speciali chiamato Program counter (PC), il cui compito e' quello di contenere l'indirizzo della successiva istruzione da passare al microprocessore.

Se pensiamo all'ufficio postale, ogni casella contiene o un'istruzione (opcode - codice operativo) o il dato/indirizzo con cui operare (operando), come mostra il programma:



Per far funzionare il nostro programma postale dovremmo scorrere ogni tabella e operare con i dati all'interno di essa. Immaginiamo un grande contatore che ci mostri l'indirizzo della casella in cui dobbiamo cercare. Normalmente questo contatore avanza di un byte alla volta; comunque, se volesse farci spostare ad un'altra area di caselle, ci indicherebbe l'indirizzo della prossima istruzione da cercare. Questo e' esattamente quello che succede con il comando JMP.

Memorizzazione nel program counter

L'istruzione JMP $indirizzo mette nel program counter l'indirizzo da due byte $indirizzo, carica la successiva

istruzione dalla memoria a quell'indirizzo e poi esegue il salto.

NOTA: le istruzioni di salto del tipo "branch" sommano o sottraggono dal program counter in maniera simile, facendo un salto percio' detto "relativo".

Il program counter e i sottoprogrammi

Se fosse possibile memorizzare il program counter prima che faccia un salto e si modifichi con il nuovo indirizzo, dopo saremmo capaci di ritornare allo stesso punto della memoria ricaricando nel program counter il pezzo di memoria che abbiamo memorizzato. In altre parole, se ci siamo accorti che il contatore dell'ufficio postale sta per cambiare, ci annotiamo l'indirizzo che esso mostra prima del cambio (indirizzo corrente), per poi rimetterlo nel contatore quando vogliamo e ritornare cosi' al punto da cui siamo partiti.

Cio' e' quello che fa una struttura con sottoprogrammi. Esempio:

```
 10 PRINT "CIAO MONDO"
 20 GOSUB 100
 30 PRINT "TUTTO BENE!"
 40 END
100 PRINT "COME VA?"
110 RETURN
```

che scrive:

```
CIAO MONDO
COME VA?
TUTTO BENE!
```

Abbiamo detto all'inizio del libro che un programma in L.M. puo' essere pensato come un sottoprogramma chiamato dal BASIC usando il comando SYS.

Naturalmente potete fare dei sottoprogrammi, all'interno di un programma in L.M., che vengono chiamati con JSR (Jump to subroutine - Salta al sottoprogramma). Come per il BASIC, per tornare indietro, ci vuole un comando che, in L.M., e' RTS (Return from subroutine - Ritorna dal sottoprogramma).

Scrivete il seguente programma:

```
NEW
NWATCH
 10 A901          LDA #$01
 20 8D15D0        STA $D015
 30 8DF807        STA $07F8
 40 A932          LDA #$32
 50 8D00D0        STA $D000
 60 8D01D1        STA $D001
 70 20L110        JSR L110
 80 C9FA          CMP #$FA
 90 D0L70         BNE L70
100 60            RTS
110 EE01D0        INC $D001
120 AD00D0        LDA $D000
130 18            CLC          - sottoprogramma
140 6904          ADC #$04
150 8D00D0        STA $D000
160 60            RTS
ENTER
RUN
```

Ricordate che questi programmi girano molto velocemente, per cui, se vi sembrera' che lo sprite non si muova, date piu' volte RUN.

L'uso dei sottoprogrammi e' indicativo di un ottimo stile di programmazione per due semplici motivi. Primo, perche' e' piu' semplice localizzare gli errori all'interno dei sottoprogrammi, potendoli provare indipendentemente dal resto del programma. Secondo, perche' usando i sottoprogrammi si puo' creare una biblioteca di importanti routine (esempio: per animare gli sprite, pulire lo schermo, ricercare byte ecc.) che poi possono essere usate da piu' programmi.

Abbiamo detto che l'indirizzo di ritorno da un sottoprogramma viene memorizzato, ma non abbiamo detto come. A noi serve un sistema di archiviazione che abbia determinati requisiti.

La struttura di controllo dello Stack

Per prima cosa deve essere flessibile e semplice da usare; secondariamente deve avere la possibilita' di poter chiamare un sottoprogramma all'interno di un altro (questo all'interno di un altro e cosi via), in modo che si abbia un sistema che sappia non solo l'indirizzo di ritorno di

tutti i sottoprogrammi, ma che sappia correttamente qual e' l'indirizzo di ritorno di ogni programma. Il sistema che viene usato per memorizzare gli indirizzi di una struttura dati e' detto "stack"; lo stack e' una struttura LIFO (Last input firsth output - Ultimo inserito primo ad uscire). Quando incontriamo un'istruzione RTS, vogliamo l'ultimo indirizzo messo nello stack da usare come indirizzo di ritorno dal sottoprogramma.

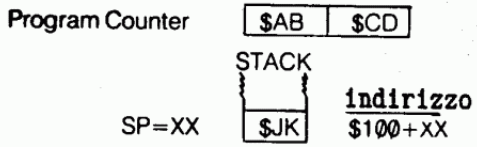
Immaginate che lo stack sia una spilla dove a volte la gente fissa dei messaggi.

Ogni volta che vedete un'istruzione JSR copiatevi l'indirizzo di ritorno su un foglio di carta e mettetelo nello stack; ripetete il processo con le prossime istruzioni JSR incontrate. Quando incontrate un'istruzione RTS, il solo pezzo di carta che potete rimuovere dalla spilla (stack) e' quello piu' in alto, che blocca tutti gli altri. Questo pezzo di carta piu' in alto conterra' sempre l'indirizzo di ritorno corretto del sottoprogramma da cui state tornando (che e' anche quello chiamato piu' recentemente).

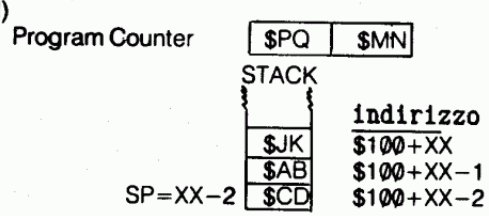
Sottoprogrammi e Stack

I comandi JSR e RTS operano automaticamente usando lo stack, che si trova in memoria a partire dall'indirizzo $100 per finire all'indirizzo $1FF (pagina 1) e si espande verso il basso (immaginate una spilla capovolta); cio' non fa alcuna differenza per il suo modo di operare. La cima dello stack (o attualmente il fondo) e' puntata da un registro speciale del microprocessore detto stack pointer (SP). Quando viene fatto un JSR, i due byte del program counter sono messi nello stack e lo stack pointer (SP) si decrementa di 2.

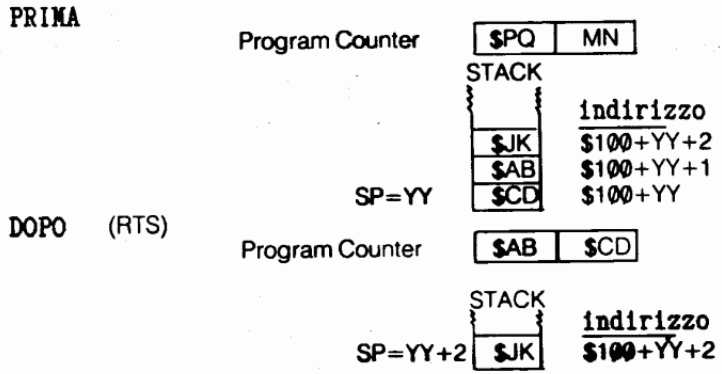
PRIMA



DOPO (JSR $PQMN)



Un RTS prende i due byte piu' alti dello stack e li passa al program counter, mentre lo stack pointer viene incrementato di 2.



Per avere un'idea della memoria dello stack fate DUMP fra gli indirizzi $100 e $200.

Mettete queste linee nel programma precedente per avere un sottoprogramma che ne chiama un altro:

```
125 20L170          JSR L170
170 C950            CMP #$50
180 90L195          BCC L195
190 EE01D0          INC $D001
195 60              RTS
ENTER
RUN
```

Uno dei piu' grossi vantaggi dello stack e' quello di poterlo usare per mettervi il contenuto dell'accumulatore o viceversa, con le istruzioni PHA (Push accumulator onto the stack - Metti l'accumulatore nello stack) e PLA (Pull accumulator off the stack - Metti lo stack in accumulatore).

ATTENZIONE: siate sicuri di prendere o mettere le cose nello stack nel giusto ordine.

Se il microprocessore incontra un'istruzione RTS quando c'e' sulla cima dello stack un dato sbagliato, andra' all'indirizzo formato dai primi due byte dello stack, qualunque essi siano.

Usate queste istruzioni per controllare le operazioni sullo stack.

```
NEW
WATCH                (indirizzo? 334)
10 20L40             JSR L40
 20 EE20D0           INC $D020
 30 60               RTS
 40 68               PLA
 50 AA               TAX
 60 68               PLA
 70 8E3403           STX $0334
 80 8D3503           STA $0335
 90 48               PHA
100 8A               TXA
110 48               PHA
120 60               RTS
```

Riepilogo del programma

Linea 10	JSR - indirizzo di ritorno (l'indirizzo della prossima istruzione e' nello stack). Attualmente punta al byte prima della prossima istruzione, perche' il PC e' incrementato sempre prima che un byte sia "prelevato" dalla memoria.
Linea 20	Incrementa il colore del bordo dello schermo (vedi appendice 6) tanto per far vedere che il programma e' ritornato correttamente.
Linea 30	FINE.
Linee 40 - 60	Prendono i due byte in cima allo stack.
Linee 70 - 80	Li memorizzano come byte basso - byte alto in $334 e $335.
Linee 90 - 110	Rimettono i byte nello stack nell'ordine giusto.
Linea 120	FINE del sottoprogramma.

Fate ENTER e RUN, controllate con WATCH l'indirizzo $335 e poi ridate RUN. Mettete insieme i risultati e confrontateli con gli indirizzi che vi aspettavate.

Per manipolazioni dirette dello stack sono possibili due istruzioni: TSX (Transfer SP into X - Metti SP in X) e TXS (Transfer X into SP - Metti X in SP). Scrivete un programma con due sottoprogrammi, uno dentro l'altro, e memorizzate lo SP in memoria (tramite X), per vedere i mutamenti nello

stack pointer, allorquando viene chiamato un sottoprogramma o viene eseguito un RTS.

Lo stack e le interruzioni

Nel capitolo 9 abbiamo menzionato il comando BRK, che viene utilizzato per la messa a punto dei programmi; bloccando questi, da' la possibilita' di esaminare le variabili in uso, a meta' del loro compito. Quello che succede con un comando BRK e' pressappoco quello che avviene con JSR; infatti BRK esegue un JSR indiretto a $FFFE, $FFFF. In altre parole, il contenuto di questi byte viene messo nel PC ed il programma continua da la' (alle routine di interruzione della ROM); BRK salva sullo stack anche il registro dei codici dello stato dei processi (P).

Quest'ultima operazione puo' essere fatta anche senza un BRK, con le istruzioni PHP (Push processor status byte - Preleva il byte dello stato dei processi) e PLP (Pull processor status byte - Tira fuori il byte dello stato dei processi). Tutto cio' permette l'accesso ad un'area molto grande di programmazione in L.M. sul Commodore 64: le Interrupts (interruzioni). Comunque, per quello che ci riguarda, possiamo dire che in questo libro non ne parleremo, perche' si tratta di un argomento troppo avanzato, ma ci limiteremo a dire come, dove e perche' esse operano. In generale, dall'hardware del calcolatore viene inviata un'interruzione al microprocessore per segnalargli quando capita qualcosa nel mondo esterno che richiede la sua attenzione (esempio: un tasto e' stato premuto).

Queste interruzioni sono segnali hardware che interrompono il microprocessore, qualsiasi cosa stia facendo, e saltano ad una routine di interruzione di servizio (tramite i vettori $FFFE e $FFFF).

In maniera simile a BRK, un'interruzione mette il PC e il registro di stato (P) nello stack e poi fa un salto indiretto al contenuto delle locazioni $FFFE e $FFFF, le quali mandano ad una routine di interruzione in ROM.

E' possibile controllare le routine di interruzioni di servizio per operare interruzioni all'orologio o ad altre sorgenti, in modo da poter fare diverse cose: muovere oggetti a velocita' costante e predefinita, incrementare l'ora dell'orologio, o altre ancora. Alcuni metodi sono descritti nel prossimo capitolo.

Premete contemporaneamete RUN/STOP e RESTORE per ritornare al video normale e poi fate GOTO 2020.

Capitolo 11 SOMMARIO

1. Il Program counter (PC) punta al prossimo byte in memoria da usare come istruzione.
2. JMP mette gli indirizzi nel PC
3. I salti "branch" sommano o sottraggono dal PC.
4. JSR mette il PC nello stack e mette nel PC il nuovo indirizzo (sottoprogramma).
5. RTS prende i due byte in cima allo stack e li mette nel PC (indirizzo di ritorno).
6. Nello stack le cose possono essere inserite solo da una parte e, solo da essa possono essere riprese, nello stesso ordine in cui sono state inserite.
7. Lo stack pointer segnala la cima dello stack.

   RTS ⇒ SP=SP+2
   JSR ⇒ SP=SP−2

8. PHA e PLA, mettono e prendono l'accumulatore dallo stack. Fate attenzione a riprendere le cose nell'ordine in cui le avete messe.
9. TXS e TSX trasferiscono i contenuti fra lo stack pointer e X.
10. BRK

```
                PC  --> Stack (2 byte)
Byte di stato  --> Stack
Contenuto di
($FFFE, $FFFF)  --> PC
```

11. PHP e PLP mettono e levano il byte dello stato dei processi dallo stack.
12. Le interruzioni arrivano da chip esterni al microprocessore

```
                PC  --> Stack (2 byte)
Byte di stato  --> Stack
($FFFE, $FFFF)  --> PC
```

Sono attivate dalla routine ROM che opera le interruzioni.

CAPITOLO 12

LAVORARE CON IL SISTEMA OPERATIVO

Il Kernal

In questo capitolo parleremo delle cose che si possono fare lavorando con il sistema operativo del Commodore 64. Il sitema operativo e' un programma residente in ROM, chiamato anche Kernal; esso si trova in memoria dagli indirizzi $E000 a $FFFF e opera con la parte hardware del calcolatore (altre ROM operano con il BASIC o i caratteri, ecc.). Nel Kernal ci sono routine per aprire o chiudere un file, per stampare caratteri sullo schermo, per leggere un carattere dalla tastiera, per muovere il cursore sullo schermo, per caricare o memorizzare file e per altre funzioni accessorie, ma non per questo meno importanti.

In questo capitolo daremo esempi su come usare alcune di queste routine. Appresi questi metodi e le informazioni date nelle appendici, potrete scrivere programmi che dialogano semplicemente e in maniera efficiente con il mondo esterno (periferiche, ecc.). Il maggiore uso del kernal e' quello concernente le interruzioni; queste possono essere mandate tanto alle periferiche, al chip sonoro, all'orologio, che in molti altri posti. L'orologio manda un'interruzione ogni 1/50 di secondo (1/60 in U.S.A), che viene usata dal Kernal per aggiornare l'orologio giornaliero e per controllare se sono stati premuti tasti sulla tastiera.

Abbiamo detto nel capitolo precedente che un'interruzione mette gli indirizzi di ritorno e il byte di stato nello stack e fa un salto indiretto ai contenuti delle locazioni $FFFE e $FFFF; abbiamo anche detto che il salto era diretto al sistema di gestione delle interruzioni in ROM. Questa routine in ROM fa il suo lavoro, ma permette anche l'accesso del programmatore ai processi di interruzione, facendo un salto ai vettori delle interruzioni che si trovano in RAM alle locazioni $314 e $315 (solito formato byte basso - byte alto), che, quindi, possono essere modificati dal nostro programma.

La nostra interruzione deve fare una delle due cose: o ritornare tramite il sistema operativo, quando ha finito il suo compito (cioe' tramite gli indirizzi che aveva nel vettore delle interruzioni prima che noi li cambiassimo),

oppure dobbiamo resettare il sistema e tornare correttamente da un'interruzione. In pratica viene scelta la prima operazione. Se vogliamo fare da soli, il nostro programma deve finire cosi':

1. Prendere i registri dallo stack (quando viene chiamata la routine di interruzione in ROM, essa mette tutti i registri nello stack, che quindi debbono poi essere rimessi a posto nello stesso ordine).

2. Riattivare le interruzioni. La routine in ROM fa un SEI che attiva il flag di interrupt, disattivando le interruzioni alla fonte.

3. Fare un RTI (Return from interrupt - ritorna da un'interruzione).

Date un'occhiata alla routine di uscita da un'interruzione in ROM, disassemblando a partire da $EA31 in giu'.

NOTA: SEI (Set interrupt flag - Attiva il flag di interrupt) non permette al microprocessore di ricevere altre interruzioni. Questa istruzione e' messa all'inizio di tutte le routine di interruzione, per essere sicuri che ogni interruzione critica venga completata prima che ne inizi un'altra.

CLI (Clear interrupt flag - Disattiva il flag di interrupt)

Riattiva la possibilita' di interruzioni; e' usata alla fine di ogni interruzione, o se quest'ultima non e' critica.

RTI (return from interrupt - Ritorna da un'interruzione)

Come l'istruzione RTS, questa rimuove tutto cio' che e' stato messo nello stack dall'interruzione (byte di stato, program counter), allo scopo di ritornare dove il programma era stato interrotto (con il byte di stato inalterato). Essa, rimettendo a posto il byte di stato, disattiva il flag di interrupt (non viene alterato quando avviene un'interruzione).

Il programma che segue e' diviso in due parti: la prima parte prepara i vettori delle interruzioni alle locazioni $314 e $315 (viene chiamata quando viene dato il RUN); la

seconda, che e' puntata dai vettori modificati, viene chiamata 50 volte ogni secondo (ogni volta che l'orologio manda un'interruzione).

```
NWATCH
NEW
10 A94C          LDA #$4C
12 8DF0CF        STA $CFF0
14 A949          LDA #$49
16 8DF1CF        STA $CFF1
18 A953          LDA #$53
20 8DF2CF        STA $CFF2
22 A954          LDA #$54
24 8DF3CF        STA $CFF3
26 A90D          LDA #$0D
28 8DF4CF        STA $CFF4
30 A900          LDA #$00
32 8DF5CF        STA $CFF5
34 78            SEI
36 A92B          LDA #$2B
38 8D1403        STA $0314
40 A9C0          LDA #$C0
42 8D1503        STA $0315
44 58            CLI
46 60            RTS

60 209FFF        JSR $FF9F
62 A6C6          LDX $C6
64 F0L90         BEQ L90
66 CA            DEX
68 BD7702        LDA $0227,X
70 C985          CMP #$85
72 D0L90         BNE L90
74 A0FF          LDY #$FF
78 C8            INY
80 E8            INX
82 B9F0CF        LDA $CFF0,Y
84 9D7702        STA $0277,X
86 D0L78         BNE L78
88 86C6          STX $C6
90 4C31EA        JMP $EA31
```

Riepilogo del programma

Il programma abilita il tasto funzione F1 a stampare LIST e un RETURN automatico, ogni volta che viene premuto. Cio' puo' essere fatto con qualsiasi messaggio e tasto

permettendovi di scrivere automaticamente i comandi piu' comunemente usati.

PARTE 1

Linee 10 - 46	Inizializzazione.
Linee 10 - 32	Preparano in memoria, a partire da $CFF0, la stringa di caratteri LIST e RETURN automatico; uno zero indica che la stringa e' finita.
Linea 34	Disabilita le interruzioni mettendo a zero il flag di interrupt. Cio' deve essere fatto cambiando i vettori delle interruzioni.
Linee 36 - 42	Modificano i suddetti vettori facendoli puntare alla Parte 2 del programma.
Linee 44 - 46	Riabilitano le interruzioni e FINE.

PARTE 2

Linee 60 - 90	Questa parte viene messa a $C02B e viene chiamata 50 volte al secondo tramite i vettori delle interruzioni.
Linea 60	Chiama la routine che scandisce la tastiera e ritorna con i tasti premuti nel buffer di tastiera a partire da $227.
Linea 62	Carica in X la lunghezza del buffer di tastiera (punta al prossimo spazio libero nel buffer).
Linea 64	Ritorna se il buffer e' vuoto.
Linea 66	Mette il puntatore al prossimo inserimento nel buffer.
Linee 68 - 72	Caricano il prossimo carattere e ritornano se non e' F1.
Linee 74 - 86	Mettono nel buffer la stringa ("LIST e RETURN").
Linea 88	Memorizza il nuovo puntatore alla lunghezza del buffer.
Linea 90	Ritorna tramite il sistema di gestione delle interruzioni in ROM.

Fate ENTER e RUN.

Il programma tornerebbe senza fare nulla, allora premete F1 e avrete la lista del programma.

Le tabelle presenti nelle appendici non sono intese a spiegare tutte le routine del Kernal in ROM dettagliatamente (la cosa richiederebbe un testo apposito), ma danno solo ed esclusivamente gli indirizzi di partenza,

alcuni requisiti per il funzionamento e le azioni che svolgono alcune delle routine piu' usate. Un modo per saperne di piu' e' quello di disassemblarle cosi' come sono messe in memoria.

BUONA FORTUNA!

C'e' ancora un'istruzione di cui non abbiamo parlato ed e' NOP (No operation - Nessuna operazione): sebbene non faccia nulla, viene usata per posticipare dei cicli a tempo, o per riempire parti di programma che volete poi cancellare. Il suo codice macchina e' $EA.

Capitolo 12 SOMMARIO

1. Il Kernal in ROM permette il contatto del computer con il mondo esterno.
2. Il Kernal risiede in memoria dagli indirizzi $E000 a $FFFF.
3. SEI - Attiva il flag di interrupt ed esclude l'intervento di ogni altra interruzione.
4. CLI - Disattiva il flag di interrupt, riabilita le interruzioni.
5. RTI - Ritorna da un'interruzione.

STACK --> Byte di stato
STACK --> PC (2 byte)

6. NOP - nessuna operazione.

APPENDICI

Introduzione alle appendici

Vi abbiamo fornito tabelle e diagrammi di informazioni per la programmazione in L.M. sul Commodore 64, che saranno sempre un utile riferimento anche quando non sarete piu' dei principianti. Abbiamo presentato spiegazioni e, talvolta, esempi sull'uso delle parti piu' utili delle tabelle. Quelli che non sono accompagnati da spiegazione vanno oltre lo scopo di questo libro e vi sono inclusi a puro titolo informativo, per fornirvi un utile riferimento e un avvio verso piu' complesse e intricate programmazioni.

APPENDICE 1

Codici delle istruzioni del 6510

Mentre scrivete programmi in L.M. dovete per forza far costante riferimento a queste tabelle, che sono un elenco di tutte le istruzioni con la descrizione, i modi di indirizzamento possibili, il formato, il numero di byte usati, il codice esadecimale e l'elenco dei flag di stato che cambiano con il risultato dell'istruzione stessa.

Istruzioni del Microprocessore 6510 in ordine alfabetico

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right one Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

Codice istruzioni del 6510

Nome Descrizione	Metodo di indirizzamento	Assembler	Numero byte	Cod. OP ESA	Registro di stato
ADC Add memory to accumulator with carry					N V - B D I Z C
	Immediate	ADC #Oper	2	69	• • • •
	Zero Page	ADC Oper	2	65	
	Zero Page.X	ADC Oper.X	2	75	
	Absolute	ADC Oper	3	6D	
	Absolute.X	ADC Oper.X	3	7D	
	Absolute.Y	ADC Oper.Y	3	79	
	(Indirect.X)	AND (Oper.X)	2	61	
	(Indirect).Y	ADC (Oper).Y	2	71	
AND "AND" memory with accumulator					N V - B D I Z C
	Immediate	AND #Oper	2	29	• •
	Zero Page	AND Oper	2	25	
	Zero Page.X	AND Oper.X	2	35	
	Absolute	AND Oper	3	2D	
	Absolute.X	AND Oper.X	3	3D	
	Absolute.Y	AND Oper.Y	3	39	
	(Indirect.X)	AND (Oper.X)	2	31	
	(Indirect).Y	AND (Oper.)Y	2	31	
ASL Shift left one bit (Memory or Accumulator) C←7 6 5 4 3 2 1 0←0					N V - B D I Z C
	Accumulator	ASL A	1	0A	• • •
	Zero Page	ASL Oper	2	06	
	Zero Page.X	ASL Oper.X	2	16	
	Absolute	ASL Oper	3	0E	
	Absolute.X	ASL Oper.X	3	1E	
BCC Branch on carry clear	Relative	BBC Oper	2	90	N V - B D I Z C
BCS Branch on carry set	Relative	BCS Oper	2	B0	N V - B D I Z C
BEQ Branch on result zero	Relative	BEQ Oper	2	F0	N V - B D I Z C
BIT Test bits in memory with accumulator					N V - B D I Z C
	Zero Page	BIT Oper	1	24	M M •
	Absolute	BIT Oper	3	2C	7 6
BMI Branch on result minus	Relative	BMI Oper	2	30	N V - B D I Z C
BNE Branch on result not zero	Relative	BNE Oper	2	D0	N V - B D I Z C
BPL Branch on result plus	Relative	BPL oper	2	10	N V - B D I Z C
BRK Force Break	Implied	BRK	1	• 00	N V - B D I Z C 1 1
BVC Branch on overflow clear	Relative	BVC Oper	2	50	N V - B D I Z C

Codice istruzioni del 6510

Nome Descrizione	Metodo di indirizzamento	Assembler	Numero byte	Cod. OP ESA	Registro di stato
BVS Branch on overflow set	Relative	BVS Oper	2	70	N V - B D I Z C
CLC Clear carry flag	Implied	CLC	1	18	N V - B D I Z C 0
CLD Clear decimal mode	Implied	CLD	1	D8	N V - B D I Z C 0
CLI Clear interrupt flag	Implied	CLI	1	58	N V - B D I Z C 0
CLV Clear overflow flag	Implied	CLV	1	B8	N V - B D I Z C 0
CMP Compare memory and accumulator	Immediate Zero Page Zero Page.X Absolute Absolute.X Absolute.Y (Indirect.X) (Indirect).Y	CMP #Oper CMP Oper CMP Oper.X CMP Oper CMP Oper.X CMP Oper.Y CMP (Oper.X) CMP (Oper).Y	2 2 2 3 3 3 2 2	C9 C5 D5 CD DD D9 C1 D1	N V - B D I Z C • • •
CPX Compare memory and index X	Immediate Zero Page Absolute	CPX #Oper CPX Oper CPX Oper	2 2 3	E0 E4 EC	N V - B D I Z C • • •
CPY Compare memory and index Y	Immediate Zero Page Absolute	CPY #Oper CPY Oper CPY Oper	2 2 3	C0 C4 CC	N V - B D I Z C • • •
DEC Decrement memory by one	Zero Page Zero Page.X Absolute Absolute.X	DEC Oper DEC Oper.X DEC Oper DEC Oper.X	2 2 3 3	C6 D6 CE DE	N V - B D I Z C • •
DEX Decrement index X by one	Implied	DEX	1	DA	N V - B D I Z C • •
DEY Decrement index Y by one	Implied	DEY	1	88	N V - B D I Z C • •

Codice istruzioni del 6510

Nome Descrizione	Metodo di indirizzamento	Assembler	Numero byte	Cod. OP ESA	Registro di stato
EOR					N V - B D I Z C
"Exclusive Or" memory with accumulator	Immediate	EOR #Oper	2	49	• •
	Zero Page	EOR Oper	2	45	
	Zero Page.X	EOR Oper.X	2	55	
	Absolute	EOR Oper	3	4D	
	Absolute.X	EOR Oper.X	3	5D	
	Absolute.Y	EOR Oper.Y	3	59	
	(Indirect.X)	EOR (Oper.X)	2	41	
	(Indirect).Y	EOR (Oper).Y	2	51	
INC					N V - B D I Z C
Increment memory by one	Zero Page	INC. Oper	2	E6	• •
	Zero Page.X	INC Oper.X	2	F6	
	Absolute	INC Oper	3	EE	
	Absolute.X	INC Oper.X	3	FE	
INX					N V - B D I Z C
Increment index X by one	Implied	INX	1	E8	• •
INY					N V - B D I Z C
Increment index Y by one	Implied	INY	1	C8	• •
JMP					N V - B D I Z C
Jump to new location	Absolute	JMP Oper	3	4C	
	Indirect	JMP (Oper)	3	6C	
JSR					N V - B D I Z C
Jump to new location saving return address	Absolute	JSR Oper	3	20	
LDA					N V - B D I Z C
Load accumulator with memory	Immediate	LDA #Oper	2	A9	• •
	Zero Page	LDA Oper	2	A5	
	Zero Page.X	LDA Oper.X	2	B5	
	Absolute	LDA Oper	3	AD	
	Absolute.X	LDA Oper.X	3	BD	
	Absolute.Y	LDA Oper.Y	3	B9	
	(Indirect.X)	LDA (Oper.X)	2	A1	
	(Indirect).Y	LDA (Oper).Y	2	B1	
LDX					N V - B D I Z C
Load index X with memory	Immediate	LDX #Oper	2	A2	• •
	Zero Page	LDX Oper	2	A6	
	Zero Page.Y	LDX Oper.Y	2	B6	
	Absolute	LDX Oper	3	AE	
	Absolute.Y	LDX Oper.Y	3	BE	
LDY					N V - B D I Z C
Load index Y with memory	Immediate	LDY #Oper	2	A0	• •
	Zero Page	LDY Oper	2	A4	
	Zero Page.X	LDY Oper.X	2	B4	
	Absolute	LDY Oper	3	AC	
	Absolute.X	LDY Oper.X	3	BC	

Codice istruzioni del 6510

Nome Descrizione	Metodo di indirizzamento	Assembler	Numero byte	Cod. OP ESA	Registro di stato
LSR Shift right one bit (memory or accumulator) 0→7 8 6 4 3 2 1 0→C	Accumulator Zero Page Zero Page.X Absolute Absolute.X	LSR A LSR Oper LSR Oper.X LSR Oper LSR Oper.X	1 2 2 3 3	4A 46 56 4E 5E	N V - B D I Z C 0 • •
NOP No operation	Implied	NOP	1	EA	N V - B D I Z C
ORA "OR" memory with accumulator	Immediate Zero Page Zero Page.X Absolute Absolute.X Absolute.Y (Indirect.X) (Indirect).Y	ORA #Oper ORA Oper ORA Oper.X ORA Oper ORA Oper.X ORA Oper.Y ORA (Oper.X) ORA (Oper).Y	2 2 2 3 3 3 2 2	09 05 15 0D 1D 19 01 11	N V - B D I Z C • •
PHA Push accumulator on stack	Implied	PHA	1	48	N V - B D I Z C
PHP Push processor status on stack	Implied	PHP	1	08	N V - B D I Z C
PLA Pull accumulator from stack	Implied	PLA	1	68	N V - B D I Z C • •
PLP Pull processor status from stack	Implied	PLP	1	28	N V - B D I Z C • • • • • • • •
ROL Rotate one bit left (memory or accumulator) 7 6 5 4 3 2 1 0←C	Accumulator Zero Page Zero Page.X Absolute Absolute.X	ROL A ROL Oper ROL Oper.X ROL Oper ROL Oper.X	1 2 2 3 3	2A 26 36 2E 3E	N V - B D I Z C • • •
ROR Rotate one bit right (memory or accumulator) C→7 6 5 4 3 2 1 0	Accumulator Zero Page Zero Page.X Absolute Absolute.X	ROR A ROR Oper ROR Oper.X ROR Oper ROR Oper.X	1 2 2 3 3	6A 66 76 6E 7E	N V - B D I Z C • • •

Codice istruzioni del 6510

Nome Descrizione	Metodo di indirizzamento	Assembler	Numero byte	Cod. OP ESA	Registro di stato
RTI Return from interrupt	Implied	RTI	1	40	N V - B D I Z C • • • • • • • •
RTS Return from subroutine	Implied	RTS	1	60	N V - B D I Z C
SBC Subtract memory from accumulator with borrow	Immediate Zero Page Zero Page.X Absolute Absolute.X Absolute.Y (Indirect.X) (Indirect).Y	SBC #Oper SBC Oper SBC Oper.X SBC Oper SBC Oper.X SBC Oper.Y SBC (Oper.X) SBC (Oper).Y	2 2 2 3 3 3 2 2	E9 E5 F5 ED FD F9 E1 F1	N V - B D I Z C • • • •
SEC Set carry flag	Implied	SEC	1	38	N V - B D I Z C 1
SED Set decimal mode	Implied	SED	1	F8	N V - B D I Z C 1
SEI Set interrupt disable status	Implied	SEI	1	78	N V - B D I Z C 1
STA Store accumulator in memory	Zero Page Zero Page.X Absolute Absolute.X Absolute.Y (Indirect.X) (Indirect).Y	STA Oper STA Oper.X STA Oper STA Oper.X STA Oper.Y STA (Oper.X) STA (Oper).Y	2 2 3 3 3 2 2	85 95 8D 9D 99 81 91	N V - B D I Z C
STX Store index X in memory	Zero Page Zero Page.Y Absolute	STX Oper STX Oper.Y STX Oper	2 2 3	86 96 8E	N V - B D I Z C
STY Store index Y in memory	Zero Page Zero Page.X Absolute	STY Oper STY Oper.X STY Oper	2 2 3	84 94 8C	N V - B D I Z C
TAX Transfer accumulator to index X	Implied	TAX	1	AA	N V - B D I Z C • •
TAY Transfer accumulator to index Y	Implied	TAY	1	A8	N V - B D I Z C • •
TSX Transfer stack pointer to index X	Implied	TSX	1	BA	N V - B D I Z C • •

Codice istruzioni del 6510

Nome Descrizione	Metodo di indirizzamento	Assembler	Numero byte	Cod. OP ESA	Registro di stato
TXA Transfer index X to accumulator	Implied	TXA	1	BA	N V - B D I Z C • •
TXS Transfer index X to stack pointer	Implied	TXS	1	9A	N V - B D I Z C
TYA Transfer index Y to accumulator	Implied	TYA	1	98	N V - B D I Z C • •

Codici operativi del microprocessore 6510 in ordine numerico secondo il loro valore

00 — BRK
01 — ORA — (Indirect.X)
02 — ???
03 — ???
04 — ???
05 — ORA — Zero Page
06 — ASL — Zero Page
07 — ???
08 — PHP
09 — ORA — Immediate
0A — ASL — Accumulator
0B — ???
0C — ???
0D — ORA — Absolute
0E — ASL — Absolute
0F — ???
10 — BPL
11 — ORA — (Indirect).Y
12 — ???
13 — ???
14 — ???
15 — ORA — Zero Page.X
16 — ASL — Zero Page.X
17 — ???
18 — CLC
19 — ORA — Absolute.Y
1A — ???
1B — ???
1C — ???
1D — ORA — Absolute.X
1E — ASL — Absolute.X
1F — ???
20 — JSR
21 — AND — (Indirect.X)
22 — ???
23 — ???
24 — BIT — Zero Page
25 — AND — Zero Page
26 — ROL — Zero Page
27 — ???
28 — PLP
29 — AND — Immediate
2A — ROL — Accumulator
2B — ???
2F — ???
30 — BMI
31 — AND — (Indirect).Y
32 — ???
33 — ???
34 — ???
35 — AND — Zero Page.X
36 — ROL —Zero Page.X
37 — ???
38 — SEC
39 — AND — Absolute.Y
3A — ???
3B — ???
3C — ???
3D — AND — Absolute.X
3E — ROL — Absolute.X
3F — NOP
40 — RTI
41 — EOR — (Indirect.X)
42 — ???
43 — ???
44 — ???
45 — EOR — Zero Page
46 – LSR – Zero Page
47 — ???
48 — PHA
49 — EOR — Immediate
4A — LSR — Accumulator
4B — ???
4C — JMP — Absolute
4D — EOR — Absolute
4E — LSR — Absolute
4F — ???
50 — BVC
51 — EOR (Indirect).Y
52 — ???
53 — ???
54 — ???
55 — EOR — Zero Page.X
56 — LSR — Zero Page.X
57 — ???
58 — CLI
59 — EOR — Absolute.Y
5A — ???
5E — LSR — Sbsolute.X
5F — ???
60 — RTS
61 — ADC — (Indirect.X)
62 — ???
63 — ???
64 — ???
65 — ACD — Zero Page
66 — ROR — Zero Page
67 — ???
68 — PLA
69 — ADC — Immediate
6A — ROR — Accumulator
6B — ???
6C — JMP — Indirect
6D — ADC — Absolute
6E — ROR — Absolute
6F — ???
70 — BVS
71 — ADC — (Indirect).Y
72 — ???
73 — ???
74 — ???
75 — ADC — Zero Page.X
76 — ROR — Zero Page.X
77 — ???
78 — SEI
79 — ADC — Absolute.Y
7A — ???
7B — ???
7C — ???
7D — ADC — Absolute.X
7E — ROR — Absolute.X
7F — ???
80 — ???
81 — STA — (Indirect.X)
82 — ???
83 — ???
84 — STY — Zero Page
85 — STA — Zero Page
86 — STX — Zero Page
87 — ???
88 — DEY
89 — ???

2C — BIT — Absolute
2D — AND — Absolute
2E — ROL — Absolute
8D — STA — Absolute
8E — STX — Absolute
8F — ???
90 — BCC
91 — STA — (Indirect).Y
92 — ???
93 — ???
94 — STY — Zero Page.X
95 — STA — Zero Page.X
96 — STX — Zero Page.Y
97 — ???
98 — TYA
99 — STA — Absolute.Y
9A — TXS
9B — ???
9C — ???
9D — STA — Absolute.X
9E — ???
9F — ???
A0 — LDY — Immediate
A1 — LDA — (Indirect.X)
A2 — LDX — Immediate
A3 — ???
A4 — LDY — Zero Page
A5 — LDA — Zero Page
A6 — LDX — Zero Page
A7 — ???
A8 — TAY
A9 — LDA — Immediate
AA — TAX
AB — ???
AC — LDY — Absolute
AD — LDA — Absolute
AE — LDX — Absolute
AF — ???
B0 — BCS
B1 — LDA — (Indirect).Y
B2 — ???
B3 — ???

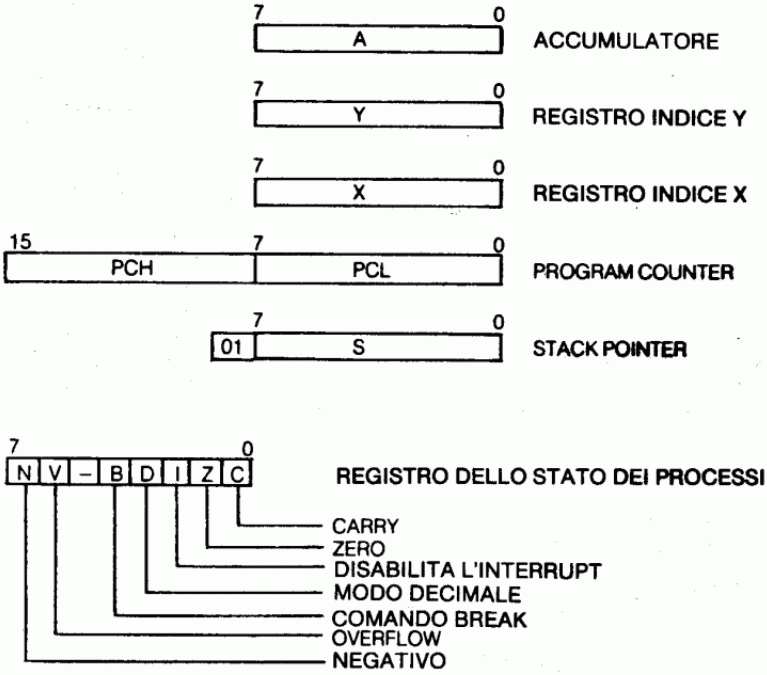
5B — ???
5C — ???
5D — EOR — Absolute.X
B4 — LDY — Zero Page.X
B5 — LDA — Zero Page.X
B6 — LDX — Zero Page. Y
B7 — ???
B8 — CLV
B9 — LDA — Absolute.Y
BA — TSX
BB — ???
BC — LDY — Absolute.X
BD — LDA — Absolute.X
BE — LDX — Absolute.Y
BF — ???
C0 — CPY — Immediate
C1 — CMP — (Indirect.X)
C2 — ???
C3 — ???
C4 — CPY — Zero Page
C5 — CMP — Zero Page
C6 — DEC — Zero Page
C7 — ???
C8 — INY
C9 — CMP — Immediate
CA — DEX
CB — ???
CC — CPY — Absolute
CD — CMP — Absolute
CE — DEC — Absolute
CF — ???
D0 — BNE
C1 — CMP — (Indirect).Y
D2 — ???
D3 — ???
D4 — ???
D5 — CMP — Zero Page.X
D6 — DEC — Zero Page.X
D7 — ???
D8 — CLD
D9 — CMP — Absolute.Y
DA — ???

8A — TXA
8B — ???
8C — STY — Absolute
DB — ???
DC — ???
DD — CMP — Absolute.X
DE — DEC — Absolute.X
DF —
E0 — CPX — Immediate
E1 — SBC — (Indirect.X)
E2 — ???
E3 — ???
E4 — CPX — Zero Page
E5 — SBC — Zero Page
E6 — INC — Zero Page
E7 — ???
E8 — INX
E9 — SBC — Immediate
EA — NOP
EB — ???
EC — CPX — Absolute
ED — SBC — Absolute
EE — INC — Absolute
EF — ???
F0 — BEQ
F1 — SBC — (Indirect).Y
F2 — ???
F3 — ???
F4 — ???
F5 — SBC — Zero Page.X
F6 — INC — Zero Page.X
F7 — ???
F8 — SED
F9 — SBC — Absolute.Y
FA — ???
FB — ???
FC — ???
FD — SBC — Absolute.X
FE — INC — Absolute.X
FF — ???

??? Operazione indefinita

APPENDICE 2

Registri del microprocessore 6510



APPENDICE 3

Tabella di conversione da esadecimale a decimale

Questa tavola si puo' usare per convertire in decimale numeri esadecimali fino a 4 cifre.

Come si usa la tavola:

1. Dividete il numero in gruppi di due cifre, ad es.

   $F17B → F1 7B
   $2A → 2A

2. Prendete il byte basso del numero (nell'esempio 7B o 2A) e cercatelo sul diagramma. Cercate la cifra piu' significativa (7) nella colonna di sinistra e la cifra meno significativa (B) sulla riga in alto, guardate dove la riga e la colonna si incrociano e troverete due numeri: 123 e 31488 . Questi sono i valori come byte basso e alto di 7B. Dato che vi serve il byte basso, prendete 123. Adesso cercate il valore per il byte alto F1, ripetete il procedimento e trovate: 241 61696 . Siccome e' il byte alto prendiamo il valore 61696 e lo sommiamo al valore trovato prima (123) per il byte basso.

```
 61696
+  123
 -----
 61819
```

NOTA: per trovare il valore decimale di un numero a due cifre, ad esempio 2A, cercatelo sul diagramma e prendete il valore del byte basso (42). Per un numero da una cifra, ad esempio E, createne uno da due aggiungendo uno zero a sinistra (0E); sempre aggiungendo uno zero creerete numeri da 3 e 4 cifre.

TABELLA DI CONVERSIONE DA ESADECIMALE A DECIMALE

CIFRA MENO SIGNIFICATIVA

CIFRA PIU' SIGNIFICATIVA

HEX	0		1		2		3		4		5		6		7		8		9		A		B		C		D		E		F	
	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte	Low Byte	High Byte
0	0	0	1	256	2	512	3	768	4	1024	5	1280	6	1536	7	1792	8	2048	9	2304	10	2560	11	2816	12	3072	13	3328	14	3584	15	3840
1	16	4096	17	4352	18	4608	19	4864	20	5120	21	5376	22	5632	23	5888	24	6144	25	6400	26	6656	27	6912	28	7168	29	7424	30	7680	31	7936
2	32	8192	33	8448	34	8704	35	8960	36	9216	37	9472	38	9728	39	9984	40	10240	41	10496	42	10752	43	11008	44	11264	45	11520	46	11776	47	12032
3	48	12288	49	12544	50	12800	51	13056	52	13312	53	13568	54	13824	55	14080	56	14336	57	14592	58	14848	59	15704	60	15360	61	15616	62	15872	63	16128
4	64	16384	65	16640	66	16896	67	17152	68	17408	69	17664	70	17920	71	18176	72	18432	73	18688	74	18944	75	19200	76	19456	77	19712	78	19968	79	20224
5	80	20480	81	20736	82	20992	83	21248	84	21504	85	21760	86	22016	87	22272	88	22528	89	22784	90	23040	91	23296	92	23552	93	23808	94	24064	95	24320
6	96	24576	97	24832	98	25088	99	25344	100	25600	101	25856	102	26112	103	26368	104	26624	105	26880	106	27136	107	27392	108	27648	109	27904	110	28160	111	28416
7	112	28672	113	28928	114	29184	115	29440	116	29696	117	29952	118	30208	119	30464	120	30720	121	30976	122	31232	123	31488	124	31744	125	32000	126	32256	127	32512
8	128	32768	129	33024	130	33280	131	33536	132	33792	133	34048	134	34304	135	34560	136	34816	137	35072	138	35328	139	35584	140	35840	141	36096	142	36352	143	36608
9	144	36864	145	37120	146	37376	147	37632	148	37888	149	38144	150	38400	151	38656	152	38912	153	39168	154	39424	155	39680	156	39936	157	40192	158	40448	159	40704
A	160	40960	161	41216	162	41472	163	41728	164	41984	165	42240	166	42496	167	42752	168	43008	169	43264	170	43520	171	43776	172	44032	173	44288	174	44544	175	44800
B	176	45056	177	45312	178	45568	179	45824	180	46080	181	46336	182	46592	183	46848	184	47104	185	47360	186	47616	187	47872	188	48128	189	48384	190	48640	191	48896
C	192	49152	193	49408	194	49664	195	49920	196	51076	197	50432	198	50688	199	50944	200	51200	201	51456	202	51712	203	51968	204	52224	205	52840	206	52736	207	52992
D	208	53248	209	53504	210	53760	211	54016	212	54272	213	54528	214	54784	215	55040	216	55296	217	55552	218	55808	219	56064	220	56320	221	56576	222	56832	223	57088
E	224	57344	225	57600	226	57856	223	58112	228	58368	229	58624	230	58880	231	59136	232	59392	233	59648	234	59904	235	60160	236	60416	237	60672	238	60928	239	61184
F	240	61440	241	61696	242	61952	243	62208	244	62464	245	62720	246	62976	247	63232	248	63488	249	63744	250	64000	251	64256	252	64512	253	64768	254	65024	255	65280

APPENDICE 4

Tabella dei salti relativi e di numerazione in complemento a due

1. Per calcolare i salti relativi, localizzate l'indirizzo immediatamente dopo la locazione dell'istruzione "branch". Contate il numero di byte da li' alla destinazione del salto; se quest'ultima viene prima del primo byte dell'istruzione di salto usate la tabella dei salti indietro, altrimenti quella dei salti in avanti. Cercate la posizione del numero contato sul corpo centrale della tabella e leggete le cifre alta e bassa del salto sui bordi Si puo' fare anche il contrario, cioe' cercare un salto sui bordi e vedere di quanti byte e' lungo guardando il corpo centrale della tabella.

2. Per convertire un numero compreso fra - 128 e + 127 in un numero esadecimale in complemento a due, cercate il numero decimale nel corpo della tabella (positivi e negativi) e leggete il numero esadecimale sui bordi (cifra alta, cifra bassa). Per fare il procedimento inverso (da esadecimale complementato a due in decimale con segno) basta cercare la cifra bassa nella colonna sulla destra e la cifra alta nella riga in alto; leggete il numero trovato dove si incontrano la colonna e la riga e, se state leggendo sulla "grafico negativo", consideratelo negativo.

Tabella dei salti Relativi
e di Numerazione in Complemento a due

SALTI RELATIVI IN AVANTI — NUMERI POSITIVI

bassa \ alta	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

SALTI RELATIVI INDIETRO — NUMERI NEGATIVI

bassa \ alta	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

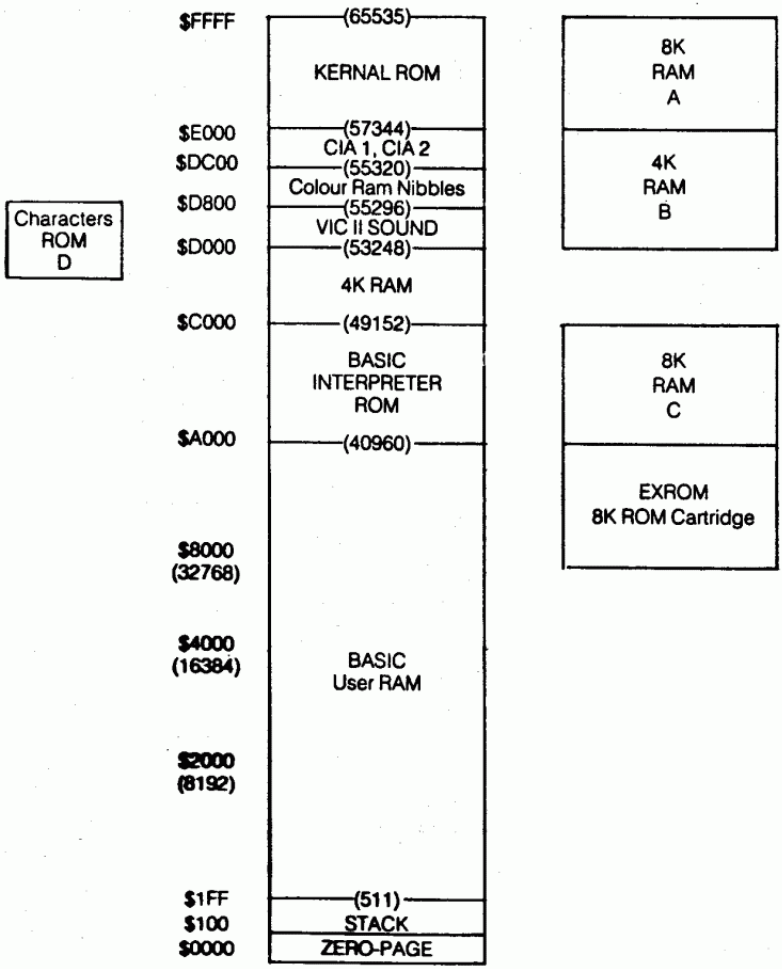
APPENDICE 5

Mappa di memoria del Commodore 64

Il Commodore64 ha 64K bytes di RAM, e sappiamo che con un indirizzo di due byte possiamo accedere a 64K. Dove sono allora le routine ROM che contengono il set di caratteri e il sistema operativo? Nel C64 ci sono, oltre ai 64K di RAM, 20K di ROM. In certe aree c'e' la RAM (o la RAM e la ROM) "mappata" sulla normale memoria: cio' significa che la memoria normale in certi posti puo' essere disinserita e sostituita da altra memoria.

L'interruttore che svolge questi scambi si trova alla locazione $0001. I primi tre bit di questa locazione sono usati per definire quale memoria e' usata, secondo la tabella allegata. Abbiamo gia' usato questi spostamenti della memoria quando abbiamo parlato della ROM contenente il generatore di caratteri (Capitolo 8). Fate riferimento a questa parte per il metodo usato.

MAPPA DI MEMORIA DEL COMMODORE 64



LOCAZIONE DI MEMORIA $0001

VALORE	FORMATO MEMORIA
xxxxx111	Normal (Not A, B, C or D)
xxxxx110	A
xxxxx101	C
xxxxx100	A, B, C
xxxxx011	D
xxxxx010	A, D
xxxxx001	C, D
xxxxx000	A, C, D (Not B)

APPENDICE 6

Il chip dello schermo

Il chip dello schermo VIC II e' un controllore del video estremamente flessibile che opera con il testo, la grafica e gli sprite del Commodore 64.

Il chip indirizza 16K di memoria; in altre parole, tutta la grafica e le informazioni della memoria colore vengono memorizzate in uno dei 4 "banchi" di memoria da 16K, che compongono la memoria del Commodore 64. Per scegliere il banco che si vuole, bisogna attivare il bit appropriato nella porta di I/O del CIA#2 all'indirizzo $DD00 (vedi appendice 8). Prima pero' la porta di direzione dei dati deve essere attivata.

L'istruzione BASIC e' la seguente:

```
POKE 56578,PEEK(56578) OR 3
```

Il comando BASIC per selezionare il banco del video e':

```
POKE 56576,(PEEK(56576) AND 252) or (3 - BANCO)
```

dove il valore di BANCO dipende da:

Valore del banco	Bit	Banco	Locazione di partenza	Intervallo del Chip VIC-II
0	00	0	$0000 0	0 – 16383 (Banco per difetto)
1	01	1	$4000 16384	16384 – 32767
2	10	2	$8000 32768	32768 – 49151
3	11	3	$C000 41952	49152 – 65535

Il chip VIC II normalmente controlla i primi 16K della memoria (Banco 0). La partenza della memoria del VIC II viene sempre riferita come indirizzo BASE.

Il chip VIC II

Registri 0 - 15 a $D000 - $D00F (53272 - 53287)
sprite 0 - 7 registri di posizione.

Ogni sprite ha due registri di posizione, X e Y, che contengono un valore dello schermo visibile, ma relativo al fascio di elettroni del quadro (Il fascio di elettroni all'interno del TV, che creano la figura - vedi il diagramma della posizione degli sprite).

E' possibile creare uno sprite che non sia visibile sullo schermo, ma che ne mantenga tutte le caratteristiche, ad esempio un test di collisione (lo vedremo piu' avanti). E' utile nei giochi in cui sono importanti le collisioni che avvengono fuori dallo schermo.

Avrete notato che il valore di Y puo' essere contenuto in un byte (e' minore di 255), ma quello di X e' piu' grande dello schermo ($158 - 334). Per questo problema guardate il Registro 16.

Registro 16 a $D010 (53288)

Contiene il bit piu' significativo di ciascuno degli 8 registri della coordinata X della posizione dello sprite. E' usato per posizioni maggiori di 255.

Registro 17 a $D011 (53289)

BIT 0 - 2 SCROLL VERTICALE

Il valore di questi bit (da 0 a 7) sposta lo schermo in su' o in giu' di un equivalente numero di pixel. Incrementando il valore di questi tre bit, e facendo uno scroll dello schermo da un estremo all'altro, e' possibile ottenere uno scroll verticale molto dolce, che fa un'ottima impressione nei programmi di giochi.

BIT 3 ALTEZZA DELLO SCHERMO

Controlla le righe di testo visualizzate (24 o 25). E' spesso usato in congiunzione con lo scroll verticale per eliminare il "gioco" prodotto da uno scroll pixel per pixel.

BIT 4 ABILITA LO SCHERMO

Controlla che qualsiasi cosa venga o no visualizzata. Puo' essere usato per cancellare lo schermo in qualsiasi momento o durante un'operazione di trasferimento di dati ad una

periferica, ad esempio durante il caricamento da cassetta. (Non cambia i dati sul video, ma evita soltanto che questi siano visualizzati finche' non e' cambiato il suo valore.)

BIT 5 MAPPA BIT

Sposta il VIC II dentro e fuori dal modo a mappa bit. Vedere avanti.

BIT 6 COLORE ESTESO

Sposta il VIC II dentro e fuori dal modo a colore esteso. Con questo modo possono essere usati colori extra sul testo o sulla grafica al prezzo di una perdita di caratteri stampabili o, alternativamente, di risoluzione orizzontale.

BIT 7 MSB REGISTRO DI QUADRO

E' il bit piu' significativo del registro di quadro (numero 18. $D012)

Registro 18 a $D012 (53290)

REGISTRO DI QUADRO

Contiene il numero della linea corrente che stiamo disegnando. I programmi in L.M. sono cosi' veloci che sviluppano migliaia di istruzioni nel tempo in cui il televisore disegna uno schermo (lo schermo viene ridisegnato 50 volte al secondo). Una scritta (memorizzazione) nel registro di quadro, provoca un'interruzione da mandare al microprocessore ogni volta che il fascio di elettroni del quadro disegna la linea specificata nella memorizzazione (se il registro dell'interrupt a $D01A e' stato attivato per abilitare le interruzioni del quadro).

Questo puo' essere usato per estendere le capacita' del VIC II nel controllare cambiamenti veloci del banco del video, modi dello schermo, sprite o dati di caratteri a

meta' schermo, aumentando di due volte o piu' (piu' interruzioni) il numero delle possibilita'.

Registri 19 - 20 a $D013 - $D014 (53291 -53292)

Penna ottica X e Y

Registro 21 a $D015 (53293)

Abilita ciascuno degli 8 sprite (0 -7) attivando il bit equivalente nel registro.

Registro 22 a $D016 (53294)

Bit 0 - 2	Scroll orizzontale (vedi verticale)
Bit 3	Larghezza dello schermo (vedi altezza)
Bit 4 - 5	Attiva e disattiva il modo multicolore simile al modo colore esteso.
Bit 6 - 7	Non usati.

Registro 23 a $D017 (53295)

Raddoppia l'ampiezza di ciascuno degli sprite 0 - 7, attivando nel registro il bit corrispondente.

Registro 24 a $D018 (53296)

Definisce la locazione della memoria di schermo, della memoria colore e di quella carattere, in conformita' al modo in cui vi trovate.
Viene spiegato piu' avanti quando tratteremo i vari modi.

Registri 25 - 26 da $D019 a $D01A (53297 - 53298)

Attivando un bit nel registro 26 viene abilitata un'interrupt; ad esempio, attivando il bit zero viene abilitata un'interrupt quando il fascio di elettroni del quadro disegna la linea specificata nel registro di quadro. Quando viene ricevuta un'interrupt e' possibile sapere con il valore del registro 25 il tipo di interrupt. Il valore letto deve essere rimesso nel registro 25 prima della prossima interrupt.

Registro 27 a $D01B (53299)

Usato per sapere quando gli sprite appaiono sia sopra che sotto gli altri dati sullo schermo.

Registro 28 a $D01C (53300)

Fa apparire gli sprite nel modo multicolore.

Registro 29 a $D01D (53301)

Espansione orizzontale (doppia). Vedi registro 23

Registri 30 - 31 da $D01E a $D01F (53302 - 53303)

Se il registro che abilita le interruzioni a $D01A e' attivato per collisioni sprite-sprite oppure per collisioni sprite-sfondo, questi registri indicano quali sprite hanno causato l'interruzione e quali no.

Registri 32 - 46 da $D020 a $D02E (53304 - 53318)

Si spiegano da soli.

Il modo carattere standard

Nel modo standard la memoria di schermo e' di 1000 byte; ciascun byte rappresenta uno dei 255 caratteri e il colore di ognuno di questi e' definito dalla RAM del colore che e' formata da 1000 locazioni a partire da $D800 a $DBE8 (55296 - 56295). Ogni pezzo di memoria nella RAM del colore e' di 4 bit e puo' memorizzare solo numeri da 0 a 15 (vedi tabella dei colori).

Sia la memoria di schermo che la memoria carattere possono essere definite in qualsiasi punto all'interno del banco video da 16K in uso, mettendo il valore appropriato nel registro 24. L'indirizzo della memoria di schermo viene definito dai bit piu' alti (7 - 4) di questo registro. Quattro bit specificano un numero da 0 a 16, cosi':

Indirizzo memoria di schermo = Indirizzo base + 1024 per quattro bit piu' significativi del registro 24.

Cambiando l'indirizzo della memoria carattere con un indirizzo in RAM, possiamo ridefinire il set di caratteri (vedere capitolo 8 per i dettagli sul set di caratteri). I

bit 3 - 1 definiscono la locazione del set di caratteri che e' un numero da tre bit (0 - 7), percio':

Indirizzo memoria carattere = Indirizzo base + 1024 per bit da 3 - 1 del registro 24.

Per accedere al set di caratteri della ROM mettete i bit 3 - 1 a 010 per il primo set e a 011 per il secondo set. Il bit 0 del registro 24 viene ignorato.

Modo carattere multicolore

E' la stessa cosa del modo standard eccetto che nel modo in cui sono definiti i caratteri (vedere capitolo 8). Nel modo standard un bit equivale ad un pixel, nel modo multicolore due bit equivalgono ad un pixel, ma puo' essere aggiunto del colore extra. Quindi un carattere multicolore e' di 4X8 pixel (ogni byte formato in 8 bit e' adesso in 4 coppie di bit).

Una coppia di bit 00 visualizza il colore di sfondo 0.
Una coppia di bit 01 visualizza il colore di sfondo 1.
Una coppia di bit 10 visualizza il colore di sfondo 2.
Una coppia di bit 11 visualizza il colore dei 3 bit bassi della memoria colore.

Se attiviamo il quarto bit della memoria colore, la posizione del carattere sullo schermo multicolore rimane nella stessa posizione anche quando si e' tornati al modo caratteri standard.

Modo colore di sfondo esteso

E' simile al modo multicolore, ma invece di perdere risoluzione riduce i caratteri stampabili da 255 a 63. Cio' significa che i due bit piu' alti di ogni byte sullo schermo possono essere usati per specificare uno dei colori di sfondo dei registri del chip VIC II. Percio' ogni carattere puo' prendere uno dei 4 colori di sfondo. Il colore in rilievo rimane quello della RAM colore.

Se i due bit piu' alti del byte in memoria di schermo sono:

=00 --> colore di sfondo 0
=01 --> colore di sfondo 1
=10 --> colore di sfondo 2
=11 --> colore di sfondo 3

Modo a mappa bit

Le tecniche del modo a mappa bit le abbiamo discusse nel capitolo 8. Lo schermo mappato a bit occupa 8K di memoria e inizia dal valore definito nei bit bassi del registro 24 a $D018. I quattro bit alti definiscono un'area di memoria che viene usata come memoria colore; per ogni posizione di un carattere si possono visualizzare due colori, uno specificato dai quattro bit piu' alti della memoria colore e uno dai quattro piu' bassi.

Modo a mappa bit multicolore

Simile al modo carattere multicolore, perde in risoluzione per aumentare le capacita' del colore. Ogni byte e' anche qui diviso in coppie di bit, visualizzate come segue:

00 --> colore di sfondo 0
01 --> 4 bit piu' bassi della memoria colore
10 --> 4 bit piu' alti della memoria colore
11 --> quattro bit della RAM colore.

MAPPA DEI REGISTRI DEL VIC II

Reg.#	7	6	5	4	3	2	1	0	
0	Sprite# 0				x — position				
1	Sprite# 0				y — position				
2	Sprite# 1				x — position				
3	Sprite# 1				y — position				
4	Sprite# 2				x — position				
5	Sprite# 2				y — position				
6	Sprite# 3				x — position				
7	Sprite# 3				y — position				
8	Sprite# 4				x — position				
9	Sprite# 4				y — position				
10	Sprite# 5				x — position				
11	Sprite# 5				y — position				
12	Sprite# 6				x — position				
13	Sprite# 6				y — position				
14	Sprite# 7				x — position				
15	Sprite# 7				y — position				
16	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	MSB of x — position
17	RASTER MSB	EXTENDED Colour	Bit map	Display enable	Screen height	—	Vertical Scroll	—	Mode / y — scroll
18	Raster register								
19	Light pen — x								
20	Light pen — y								
21	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite enable
22	—	—	Reset Multi	Multi colour	Screen width	—	Horizontal scroll	—	Multi colour / x scroll
23	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite y — expand
24	— Screen location —				— Character base —			—	
25	IRQ	—	—	—	Light pen	Sprite-Sprite collision	Sprite-b gnd collision	Raster	Interrupt register
26	IRQ	—	—	—	Light pen	Sprite-Sprite collision	Sprite b gnd collision	Raster	Interupt enable
27	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite — background priority
28	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite multicolour select
29	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite x — expand
30	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite to Sprite collision
31	Sprite #7	Sprite #6	Sprite #5	Sprite #4	Sprite #3	Sprite #2	Sprite #1	Sprite #0	Sprite to background collision
32	Screen border Colour								
33	Background Colour #0								
34	Background Colour #1								
35	Background Colour #2								
36	Background Colour #3								
37	Sprite multicolour #0								
38	Sprite multicolour #1								
39	Sprite #0 Colour								
40	Sprite #1 Colour								
41	Sprite #2 Colour								
42	Sprite #3 Colour								
43	Sprite #4 Colour								
44	Sprite #5 Colour								
45	Sprite #6 Colour								
46	Sprite #7 Colour								

TABELLA DEI COLORI

Ø Nero	4 Porpora	8 Arancione	12 Grigio 2
1 Bianco	5 Verde	9 Marrone	13 Verde chiaro
2 Rosso	6 Blu	10 Rosa	14 Celeste
3 Turchese	7 Giallo	11 Grigio 1	15 Grigio 3

DIAGRAMMA MESCOLE COLORI

Colore Carattere

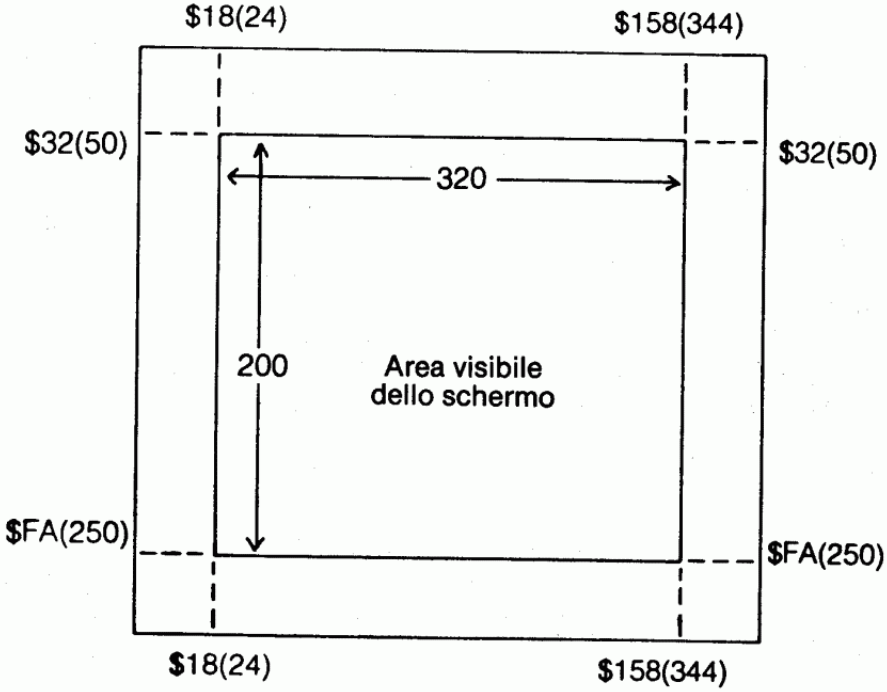
	Ø	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ø	X	✓	X	✓	✓	•	X	✓	✓	x	✓	✓	✓	✓	✓	✓
1	✓	X	✓	X	✓	✓	✓	X	•	✓	•	✓	✓	X	✓	✓
2	X	✓	X	X	•	X	X	✓	✓	X	✓	X	X	X	X	•
3	✓	X	X	X	X	•	✓	X	X	X	X	•	X	X	•	X
4	✓	•	X	X	X	X	X	X	X	X	X	X	X	X	X	•
5	✓	•	X	•	X	X	X	X	X	X	X	•	X	✓	X	•
6	•	✓	X	✓	X	X	X	X	X	X	X	X	X	•	✓	✓
7	✓	X	✓	X	X	X	•	X	•	✓	•	✓	✓	X	X	X
8	•	✓	✓	X	X	X	X	✓	X	✓	X	X	X	X	X	•
9	X	✓	X	X	X	X	X	✓	✓	X	✓	X	X	X	X	✓
10	•	•	✓	X	X	X	X	•	X	✓	X	X	X	X	X	•
11	✓	✓	X	•	X	X	X	✓	X	X	X	X	✓	✓	•	✓
12	✓	✓	•	X	X	X	•	X	X	•	X	✓	X	X	X	✓
13	✓	X	X	X	X	✓	•	X	X	X	X	✓	X	X	X	X
14	✓	✓	X	✓	X	X	✓	X	X	X	X	•	X	X	X	•
15	✓	✓	✓	X	•	•	✓	X	X	•	•	✓	✓	X	•	X

✓ Mescola buona

• Mescola normale

X Mescola pessima

DIAGRAMMA POSIZIONI SPRITE



ROM GENERATORE CARATTERI

INDIRIZZO		CONTENUTI
ESA	DECIMALE	
D000 - D1FF	53248	Maiuscole
D200 - D3FF	53760	Caratteri grafici
D400 - D5FF	54272	Maiuscole in campo inverso
D600 - D7FF	54784	Caratteri grafici in campo inverso
D800 - D9FF	55296	Minuscole
DA00 - DBFF	55808	Maiuscole e caratteri grafici
DC00 - DDFF	56320	Minuscole in campo inverso
DE00 - DFFF	56832	Maiuscole e caratteri grafici in campo inverso

NOTA: vedere l'appendice 5 per sapere come accedere a questa memoria.

APPENDICE 7

Il chip del suono

I registri del chip sonoro

REG No.	b7	b6	b5	b4	b3	b2	b1	b0	Significato del bit / Uso del registro (Voce-1)
0	NL_7	NL_6	NL_5	NL_4	NL_3	NL_2	NL_1	NL_0	Byte basso della frequenza
1	NH_7	NH_6	NH_5	NH_4	NH_3	NH_2	NH_1	NH_0	Byte alto della frequenza
2	PL_7	PL_6	PL_5	PL_4	PL_3	PL_2	PL_1	PL_0	Byte basso dell'ampiezza
3	—	—	—	—	PH_3	PH_2	PH_1	PH_0	Byte alto dell'ampiezza
4	Noise				TEST	RING MOD	SYNC	GATE	Controllo della forma d'onda
5	A_3	A_2	A_1	A_0	D_3	D_2	D_1	D_0	Attacco/ caduta per l'inviluppo
6	S_3	S_2	S_1	S_0	R_3	R_2	R_1	R_0	Sostegno/ stacco per l'inviluppo

Le voci 2 e 3 sono la copia esatta di quella sopra solo che sono memorizzate rispettivamente nei registri da 7 a 13 e da 14 a 20

									Significato del bit / Uso del registro (Filtro)
21	—	—	—	—	—	CL_2	CL_1	CL_0	Frequenza bassa di taglio
22	CH_7	CH_6	CH_5	CH_4	CH_3	CH_2	CH_1	CH_0	Frequenza alta di taglio
23	R_3	R_2	R_1	R_0	F_{EX}	F3	F2	F1	Accensione filtri e risonanza
24	3_{OFF}	HP	BP	LP	V_3	V_2	V_1	V_0	Tipo di filtro e volume.

									Significato del bit / Uso del registro (Vari)
25	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Paddle - x
26	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Paddle - y
27	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0	Oscillatore - uscita 3
28	E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0	Inviluppo - uscita 3

NOTA: I registri del chip sonoro sono accessibili tramite le locazioni di memoria da $D400 a $D41C (54272 - 54300).

Generazione di suoni

Il modo piu' pratico per scrivere programmi BASIC che generino suoni e' quella di utilizzare le istruzioni DATA. Per suoni semplici sono necessari solo due dati: la frequenza della nota e la sua durata. Per creare semplici suoni i seguenti passi sono necessari.

1. Semplificate tutte le locazioni di memoria dei registri assegnando loro delle variabili.

2. Disattivate il chip sonoro mettendo a zero tutti i registri.

3. Mettete nei registri Attacco/Caduta e Sostegno/Stacco i rispettivi valori scelti per l'inviluppo.

4. Mettete il registro del volume al massimo (15).

5. Costruite un ciclo che:
legga la frequenza e la durata della nota successiva; finisca se non ci sono piu' frequenze; altrimenti, carichi i dati nel registro della frequenza; attivi la forma d'onda e il bit GATE (vedere mappa dei registri); usi un ciclo FOR NEXT per la durata, disattivi il bit GATE; usi un ciclo FOR NEXT per creare una pausa e torni indietro per ripetere il tutto di nuovo.

6. Usate la tabella delle note ricordando che la durata 1000 corrisponde circa a 1 secondo.

7. Fate terminare le istruzioni DATA con tre valori negativi, per segnalare la fine del suono.

Il programma che dovrete scrivere dovra' essere del tipo:

```
 5 REM X SUONI X
10 CHIP=54272:C=CHIP
20 NL(0)=C+0:NH(0)=C+1:W(0)=C+4:AD(0)=C+5:
   SR(0)=C+6:VOLUME=C+24
30 FOR REG=CHIP TO CHIP+24:POKE REG,0:NEXT
40 POKE AD(0),64+9:POKE SR(0),240+0
50 POKE VOLUME,15
```

```
 60 READ F,DUR:IF F<0 THEN POKE W(0),0:END
 65 DUR=DUR*20
 70 NH(1)=INT(F/256):NL(1)=F-NH(1)*256:POKE NH(0),
    NH(1):POKE NL(0),NL(1)
 80 POKE W(0),32+1:REM SOMMA 1 A GATE
 90 FOR COUNT=1 TO DUR:NEXT COUNT
100 POKE W(0),32:REM DISATTIVA GATE
110 FOR PAUSE=1 TO 50:NEXT
120 GOTO 60
310 DATA 4820,8,6420,8,6420,12,6068,4,6420,8,8100,
    8,8100,8,7220,8,9637,8
320 DATA 9637,8,9637,12,8581,4,8100,8,7220,8,8100,
    16,4820,8,6420,8,6420,12
330 DATA 6068,4,6420,8,8100,8,8100,8,7220,8,9637,8
    ,7220,8,7220,12,6068,4
340 DATA 6068,8,5396,8,4820,16
399 DATA -1,-1,-1
```

Registri 0 e 1 (Locazioni $D400 e $D401)(54272 - 54273)
(Byte basso e alto della frequenza della nota)

Corrispondono al valore della nota suonata (nella forma a due byte). Per ottenere la frequenza della nota suonata moltiplicate i due byte per 0.059604645.

Registri 2 e 3 (Locazioni $D402 e $D403)(54274 - 54275)
(Ampiezza della cadenza)

Formano un byte da 12 bit corrispondente all'ampiezza della cadenza (pulsazione). L'ampiezza della cadenza bassa e' data come percentale dell'ampiezza del ciclo di pulsazioni e si ricava con questa formula:
Ampiezza cadenza bassa = (valore da 12 bit/40.95)% del ciclo di pulsazioni; dove un'ampiezza della cadenza bassa dello 0% o del 100% e' un segnale DC costante (es, uscita zero) e del 50% e' un'onda quadra.

Registro 4 (Locazione $D404(54276)
(Controllo forma d'onda)

Svolge svariate funzioni a seconda del bit usato.

Bit 0 (GATE):
Questo bit controlla la generazione dell'inviluppo.

Attivandolo si avvia l'inviluppo ACSS (Attacco, caduta ecc) che inizia alla fase Attacco, poi prosegue con la fase Caduta e finalmente con la fase Sostegno. Il suono rimane sempre nella fase Caduta finche' il GATE non e' messo a zero, nel qual caso il controllo dell'inviluppo passa alla fase Stacco. Se il bit e' attivato prima della fase Sostegno, il controllo passa alla fase Stacco direttamente.

Bit 1 (SYNC):
Attivare questo bit significa sincronizzare la forma d'onda della voce 3 con la voce 1. Cambiando la frequenza della voce 3 si cambia la forma d'onda della voce 1.

Bit 2 (RING MOD)
Attivando la modulazione del trillo (ring mod), viene riattivata la forma d'onda triangolare della voce 1 con una combinazione degli oscillatori 1 e 3: si ottiene cosi' un suono simile ad una campanella. Variando la frequenza dell'oscillatore 3 si provoca una variazione della forma d'onda della voce 1.

Bit 3 (TEST)
Viene usato per dei controlli; quando e' attivato, l'oscillatore 1 viene rimesso a 0 e rimane cosi' finche' il bit non viene disattivato. Molto utile per sincronizzare l'oscillatore 1 con apparecchiature esterne.

Bit 4 (Onda triangolare)
Quando attivato, abilita la forma d'onda triangolare per l'oscillatore 1.

Bit 5 (Onda a dente di sega)
Abilita la forma d'onda a dente di sega.

Bit 6 (Onda rettangolare - Pulsazione, cadenza)
Abilita la forma d'onda rettangolare.

Bit 7 (Rumore)
Abilita il rumore bianco.

Registro 5 (Locazione $D405)(54277)
(Attacco/caduta)

E' usato per selezionare la velocita' di attacco e di caduta per la voce 1 nell'inviluppo ACSS.

Bit 4 - 7 (Velocita' di attacco):

Seleziona la velocita' di attacco da 0 a 240, dove il tempo di attacco e' in un range da 2ms a 8s.

Bit 0 - 3 (velocita' di caduta):
Seleziona la velocita' di caduta da 0 a 15, dove il tempo di caduta e' in un range da 6ms a 24s.

Registro 6 (Locazione $D406)(54278)
(Sostegno/stacco)

Viene usato per scegliere il livello di sostegno e la velocita' di stacco per la voce 1 nell'inviluppo ACSS.

Bit 4 - 7 (Velocita' di sostegno):
Seleziona un livello di sostegno da 0 a 240 dove il livello di sostegno e' in proporzione al volume.
Volume di sostegno = (volume X sostegno)/240

Bit 0 - 3 (velocita' di stacco):
Seleziona la velocita' di stacco da 0 a 15 dove il tempo di stacco e' in un range da 6ms a 24s.

Registri 7 - 13 (Locazioni da D07 a $D40D)(54279 - 54285)
(Voce 2)

Identici ai registri da 0 a 6 (voce 1) con queste eccezioni:

1. SYNC - Sincronizza l'oscillatore 1 con il 2.
2. RING MOD - Ripristina la forma triangolare dell'oscillatore 2 operando una combinazione dell'oscillatore 2 con l'1.

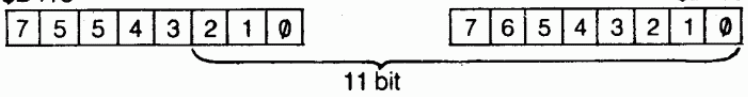
Registri 14 - 20 (Locazioni $D40E - $D414)(54286 - 54292)
(Voce 3)

Sono identici a quelli da 0 a 6 (voce 1) e a quelli da 7 a 13 (voce 2) con queste eccezioni:

1. SYNC - Sincronizza l'oscillatore 3 con il 2.
2. RING MOD - Ripristina la forma triangolare dell'oscillatore 3 con una combinazione fra il 3 e il 2.

Registri 21 e 22 (Locazioni $D415 e $D416)(54293 - 54294)

(Frequenza di chiusura)



Questi due registri, formati da un valore da 11 bit, corrispondono alla frequenza di chiusura (o centro) di un filtro programmabile. Selezionano un valore da 0 a 262, dove la frequenza di chiusura e' in un range da 30Hz a 12KHz.

Registro 23 (Locazione $D417)(54295)
(Risonanza/filtro)

Viene usato come interruttore per la risonanza e il filtro.

Bit 0 (Interruttore filtro 1):
Quando e' attivato,la voce 1 viene filtrata.

Bit 1 e 2 (Interruttore filtro 2 e 3):
Stessa cosa del bit 0, solo che vale per le voci 2 e 3.

Bit 3 (Interruttore filtro EXT):
Come il bit zero, ma vale per entrate audio da apparecchiature esterne.

Bit 4 - 7 (Risonanza):
Questi bit formano un valore da 4 bit corrispondente alla risonanza del filtro programmabile; selezionano la risonanza in un range da 16 a 240, in passi di 16. Essa opera in una piccola banda di frequenza, vicino alla frequenza di chiusura selezionata.

Registro 24 (Locazione $D418)(54296)
(Interruttore voce 3/modi filtro/volume)

Bit 0 - 3 (Volume):
Questi 4 bit selezionano il volume in un range da 0 a 15. Il registro 24 e' il principale controllo del volume. Ciascuna voce puo' essere modificata nella fase di attacco, mettendo a zero il bit GATE e scegliendo un valore di attacco molto grande; oppure selezionando un livello

differente di sostegno per ciascuna voce, il che provoca un differente livello di volume all'interno del livello assoluto determinato dai quattro bit.

Bit 4 - 6 (Modi filtro):
I tre bit sono usati per scegliere i modi del filtro programmabile. Il bit 4 attiva un filtro sui toni bassi, il 5 sui toni medi e il 6 su quelli alti. Non si puo' attivare piu' di un filtro per volta.

Bit 7 (Interruttore voce 3)
Attivato, disabilita la voce 3 senza alterarne i controlli. Questo interruttore e' molto utile quando la voce 3 e' usata per controllare un'altra voce e l'uscita della 3 non e' necessaria.

Registri 25 e 26 (Locazioni $D419 e $D41A)(54297 e 54298)
(Paddles)

Permettono al microprocessore di leggere la posizione di un paio di paddles collegate tramite la porta 1. Le paddles danno letture da 0 (resistenza minima) a 255 (resistenza massima). Leggendo questi registri e copiandoli in altri, e' possibile controllare il suono usando le paddles.

Registro 27 (Locazione 4D41B)(54299)
(Uscita oscillatore 3)

Permette al microprocessore di leggere la forma d'onda d'uscita della voce 3, dove ogni forma produce valori da 0 a 255.

Registro 28 (Locazione $D41C)(54300)
(Inviluppo uscita voce 3)

Come il registro 27, solo che permette di leggere l'inviluppo.

APPENDICE 8

Il chip interfaccia CIA (Complex Interface Adapter)

Nel Commodore 64 ci sono due chip CIA, uno usato per le funzioni di sistema e l'altro per le funzioni dell'utente. Ciascun chip (#1 e #2) ha 16 registri mappati in memoria da $DC00 a $DC0F (#1) e da $DD00 a $DD0F (#2).

Registri 0 - 3

Questi registri sono combinati per creare due registri bidirezionali di I/O. I registri 0 e 1 contengono i dati per questa I/O e i registri 2 e 3 contengono la maschera di controllo per il movimento dei dati. Se il registro 2 contiene $E0 (11100000) come maschera, sono effettivi solo i tre bit piu' alti del registro 0. (Per un'operazione di lettura (LDA Reg 0) i bit 5, 6 e 7 saranno l'input (entrata); per un'operazione di scrittura (STA Reg 0) saranno l'uscita (output). Le stesse applicazioni valgono per i registri 3 e 1.

CIA #1

	Registro 0	$DC00 (56320)
Scrittura	bit 0 - 7	Esaminano le colonne della tastiera
Lettura	bit 0 - 7	Penna ottica
	oppure	
	bit 6 - 7	Leggono le paddles
	bit 2 - 3	Tasto di sparo delle paddles
	oppure	
	bit 4	Joistick 0, tasto di sparo
	bit 0 - 3	Joistick 0, direzione
	Registro 1	$DC01 (56321)
Scrittura	bit 6 - 7	Traversino per Timer A e B
Lettura	bit 0 - 7	Esaminano le righe della tastiera
	oppure	
	bit 2 - 3	Tasto di sparo delle paddles
	oppure	
	bit 4	Joistick 1, tasto di sparo
	bit 0 - 3	Joistick 1, direzione

Tavola di direzione del joystick

Bits	0-3	Direzione
0 0	0 0	Nessuno
0 0	0 1	Sù
0 0	1 0	Giù
0 0	1 1	—
0 1	0 0	Sinistra
0 1	0 1	Sù a sinistra
0 1	1 0	Giù a destra
0 1	1 1	—
1 0	0 0	Destra
1 0	0 1	Sù a destra
1 0	1 0	Giù a destra

CIA #1

7	6	5	4	3	2	1	0
—	—	—	SPARO	DESTRA	SINISTRA	GIU'	SU'

(Bit usati dalle locazioni 56320 e 56321 per le porte joistick 1 e 2 rispettivamente).

CIA #2

	Registro 0	$DD00 (56576)
Scrittura	bit 5	Output dati bus seriale
	bit 4	Output impulsi orologio bus seriale
	bit 3	Output segnale ATN bus seriale
	bit 2	Output dati RS-232
	bit 0 - 1	Selezione memoria chip VIC II
Lettura	bit 7	Input dati bus seriale
	bit 6	Input impulsi orologio bus seriale
	Registro 1	$DD01 (56577)
	Porta utente oppure	
	bit 7 - RS-232	Posizionamento dati pronto
	bit 6 - RS-232	Prepara all'invio di nuovi dati
	bit 5 - RS-232	Controllo trasportatore
	bit 4 - RS-232	Indicatore segmento
	bit 3 - RS-232	Terminale dati pronto
	bit 2 - RS-232	Richiesta di invio
	bit 1 - RS-232	Dati ricevuti

I prossimi registri sono usati allo stesso modo sia sul CIA#1 che sul CIA#2 e sono elencati insieme.

Registri 2 - 3

Registri della direzione dei dati dei registri 0 e 1.
CIA #1 $DC02 e $DC03 (56322 - 56323) per $DC00 e $DD01
CIA #2 $DD02 e $DD03 (56578 - 56579) per $DD00 e $DD01

Registri 4 - 7		CIA#1	CIA#2
Timer A	byte basso	$DC04	$DD04
Timer A	byte alto	$DC05	$DD05
Timer B	byte basso	$DC06	$DD06
Timer B	byte alto	$DC07	$DD07

CIA #1

Questi timer sono usati per aggiornare l'orologio del BASIC, per generare le interruzioni, per leggere la tastiera e per altre funzioni.

CIA #2

Timer per le interruzioni dell'utente
Scrittura Il valore da 16 bit (byte alto, byte basso) provoca un'interrupt quando viene ricevuto dal timer corrispondente, ovviamente se i bit di abilitazione sono attivi.
Lettura Il valore corrente di 16 bit del timer.

Registri 8 - 11		CIA#1	CIA#2
Tempo dell'orologio giornaliero:	1/10 secondo	$DC08	$DD08
" :	secondi	$DC09	$DD09
" :	minuti	$DC0A	$DD0A
" :	bit 6 - 7 ora bit 7 AM/PM	$DC0B	$DD0B

CIA #1

Per il BASIC e l'orologio di sistema.

CIA #2

Scrittura Rimette l'ora.
Lettura Legge l'ora del giorno.

Registro 12	CIA#1	CIA#2
Buffer seriale dati	$DC0C	$DD0C

Registro 13	CIA#1	CIA#2
	$DC0D	$DD0D

Scrittura bit 7 Se viene messo a 1, attiva i corrispondenti bit in scrittura che sono a 1, ma non ha effetto con quelli che sono a 0.
Se viene messo a 0, attiva i corrispondenti bit in scrittura che sono a 0, ma non ha effetto con quelli che sono a 1.

bit 4 - 0 Abilita le interruzioni all'interno del chip CIA per queste aree:

Set/clear	—	—	Flag	Serial Port	Alarm	Timer B	Timer A

Lettura bit 7 Interruzione
bit 4 - 0 Segnala che una di queste sorgenti ha causato l'interruzione:

Interrupt	0	0	Flag	Serial Port	Alarm	Timer B	Timer A

CIA #1 - Flag - Soppravvenuta NMI.
CIA #2 - Flag - " IRQ.

Registri 14 - 15	CIA#1	CIA#2
Registro controllo A	$DC0E	$DD0E
Registro controllo B	$DC0F	$DD0F

Scrittura	bit 0	1 - Attiva il timer 0 - Ferma il timer
	bit 1	Timer overflow. Timer A bit 6 timer B bit 7.
	bit 2	Modo output timer al bit del registro 1.
	bit 3	1 - Una battuta (se il contatore e' a zero si ferma). 0 - Continuo (ciclo)
	bit 4	1 - Carica un valore nel timer 0 - Nessuna azione

Registro controllo A

bit 5	1 - Timer A conta segnali CNT 0 - Timer A conta segnali dell'orologio di sistema
bit 6	1 - Output porta seriale 0 - Input porta seriale
bit 7	1 - Orologio a 50Hz 0 - Orologio a 60Hz

Registro controllo A

bit 5 - 6	00 - Contatore segnali dell'orologio di sistema 01 - Contatore segnali CNT 10 - Contatore del timer A per segnali sotto livello (orologio sistema). 11 - Contatore del timer A per segnali sotto livello (CNT)

Chip interfaccia CIA
(Complex interface adapter)

Mappa registri

Reg	Name		#1 $DC00 - $DC0F Purpose	#2 $DD00 - $DD0F Purpose
0	PRA	Peripheral Data Reg A	Keyboard, Joysticks, Paddle, Light Pen	Serial Bus, RS232, Memory Control
1	PRB	Peripheral Data Reg B	Keyboard, Joysticks, Paddles	RS232, User Port
2	DDRA	Data Direction Reg A	To define which bits of A are read	To define which bits of A are read
3	DDRB	Data Direction Reg B	To define which bits of B are read	To define which bits of B are read
4	TA LO	Timer A Low Register	System Interrupt Timer	User Timer
5	TA HI	Timer A High Register	System Interrupt Timer	User Timer
6	TB LO	Timer B Low Register	System Interrupt Timer	User Timer
7	TB HI	Timer B High Register	System Interrupt Timer	User Timer
8	TOD 10THS	10THS of Seconds Register	System Clock	User Clock
9	TOD SEC	Seconds Register	System Clock	User Clock
A	TOD MIN	Minutes Register	System Clock	User Clock
B	TOD HR	Hours — AM/PM Register	System Clock	User Clock
C	SDR	Serial Data Register	Serial I/O	Serial I/O
D	ICR	Interrupt Control Register	System Interrupt Control	User Interrupt Control
E	CRA	Control Reg A	System Timer A Controller	User Timer A Controller
F	CRB	Control Reg B	System Timer B Controller	User Timer B Controller

APPENDICE 9

Uso delle locazioni di memoria

Elenco dei puntatori, buffer e variabili usate dal sistema e dal BASIC.

MAPPA DI MEMORIA DEL COMMODORE 64

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
D6510	0000	0	6510 On-Chip Data-Direction Register
R6510	0001	1	6510 On-Chip Eight-Bit Input/Output Register
	0002	2	Unused
ADRAY1	0003–0004	3–4	Jump Vector: Convert Floating—Integer
ADRAY2	0005–0006	5–6	Jump Vector: Convert Integer—Floating
CHARAC	0007	7	Search Character
ENDCHR	0008	8	Flag: Scan For Quote At End Of String
TRMPOS	0009	9	Screen Column From Last TAB
VERCK	000A	10	Flag: 0 = Load, 1 = Verify
COUNT	000B	11	Input Buffer Pointer/ Number Of Subscripts
DIMFLG	000C	12	Flag: Default Array DI-Mension
VALTYP	000D	13	Data Type: $FF = String, $00 = Numeric
INTFLG	000E	14	Data Type: $80 = Integer, $00 = Floating
GARBFL	000F	15	Flag: DATA Scan/LIST Quote/Garbage Coll
SUBFLG	0010	16	Flag: Subscript Reference User Function Call
INPFLG	0011	17	Flag: $00 = INPUT, $40 = GET, $98 = READ
TANSGN	0012	18	Flag: TAN Sign/Comparison Result
	0013	19	Flag: INPUT Prompt
LINNUM	0014–0015	20–21	Temp: Integer Value
TEMPPT	0016	22	Pointer: Temporary String Stack

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
LASTPT	0017–0018	23–24	Last Temporary String Address
TEMPST	0019–0021	25–33	Stack For Temporary Strings
INDEX	0022–0025	34–37	Utility Pointer Area
RESHO	0026–002A	38–42	Floating-Point Product Of Multiply
TXTTAB	002B–002C	43–44	Pointer: Start Of BASIC Text
VARTAB	002D–002E	45–46	Pointer: Start Of BASIC Variables
ARYTAB	002F–0030	47–48	Pointer: Start Of BASIC Arrays
STREND	0031–0032	49–50	Pointer: End Of BASIC Arrays (+1)
FRETOP	0033–0034	51–52	Pointer: Bottom Of String Storage
FRESPC	0035–0036	53–54	Utility String Pointer
MEMSIZ	0037–0038	55–56	Pointer: Highest Address Used By BASIC
CURLIN	0039–003A	57–58	Current BASIC Line Number
OLDLIN	003B–003C	59–60	Previous BASIC Line Number
OLDTXT	003D–003E	61–62	Pointer: BASIC Statement For CONT
DATLIN	003F–0040	63–64	Current DATA Line Number
DATPTR	0041–0042	65–66	Pointer: Current DATA Item Address
INPPTR	0043–0044	67–68	Vector: INPUT Routine
VARNAM	0045–0046	69–70	Current BASIC Variable Name
VARPNT	0047–0048	71–72	Pointer: Current BASIC Variable Data
FORPNT	0049–004A	73–74	Pointer: Index Variable For FOR/NEXT
	004B–0060	75–96	Temp Pointer/Data Area
FACEXP	0061	97	Floating-Point Accumulator #1: Exponent
FACHO	0062–0065	98–101	Floating Accumulator. #1: Mantissa
FACSGN	0066	102	Floating Accumulator #1: Sign
SGNFLG	0067	103	Pointer: Series Evaluation Constant
BITS	0068	104	Floating Accumulator #1: Overflow Digit

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
ARGEXP	0069	105	Floating-Point Accumulator #2: Exponent
ARGHO	006A–006D	106–109	Floating Accumulator #2: Mantissa
ARGSGN	006E	110	Floating Accumulator #2: Sign
ARISGN	006F	111	Sign Comparison Result: Accumulator #1 vs #2
FACOV	0070	112	Floating Accumulator #1. Low-Order (Rounding)
FBUFPT	0071–0072	113–114	Pointer: Cassette Buffer
CHRGET	0073–008A	115–138	Subroutine: Get Next Byte Of BASIC Text
CHRGOT	0079	121	Entry To Get Same Byte Of Text Again
TXTPTR	007A–007B	122–123	Pointer: Current Byte Of Basic Text
RNDX	008B–008F	139–143	Floating RND Function Seed Value
STATUS	0090	144	Kernal I/O Status Word: ST
STKEY	0091	145	Flag: STOP key/RVS key
SVXT	0092	146	Timing Constant For Tape
VERCK	0093	147	Flag: 0 = Load, 1 = Verify
C3PO	0094	148	Flag: Serial Bus—Output Character Buffered
BSOUR	0095	149	Buffered Character For Serial Bus
SYNO	0096	150	Cassette Sync Number
	0097	151	Temporary Data Area
LDTND	0098	152	Number Of Open Files/ Index To File Table
DFLTN	0099	153	Default Input Device (0)
DFLTO	009A	154	Default Output (CMD) Device (3)
PRTY	009B	155	Tape Character Parity
DPSW	009C	156	Flag: Tape Byte-Received
MSGFLG	009D	157	Flag: $80 = Direct Mode, $00 = Program
PTR1	009E	158	Tape Pass 1 Error Log
PTR2	009F	159	Tape Pass 2 Error Log
TIME	00A0–00A2	160–162	Real-Time Jiffy Clock (approx) 1/60th Sec
	00A3–00A4	163–164	Temporary Data Area
CNTDN	00A5	165	Cassette Sync Countdown
BUFPNT	00A6	166	Pointer: Tape I/O Buffer
INBIT	00A7	167	RS-232 Input Bits/Cassette Temp

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
BITCI	00A8	168	RS-232 Input Bit Count Cassette Temp
RINONE	00A9	169	RS-232 Flag: Check For Start Bit
RIDATA	00AA	170	RS-232 Input Byte Buffer/Cassette Temp
RIPRTY	00AB	171	RS-232 Input Parity Cassette Short Cnt
SAL	00AC–00AD	172–173	Pointer: Tape Buffer Screen Scrolling
EAL	00AE–00AF	174–175	Tape End Addresses/End Of Program
CMPO	00B0–00B1	176–177	Tape Timing Constants
TAPE1	00B2–00B3	178–179	Pointer: Start Of Tape Buffer
BITTS	00B4	180	RS-232 Out Bit Count Cassette Temp
NXTBIT	00B5	181	RS-232 Next Bit To Send Tape EOT Flag
RODATA	00B6	182	RS-232 Out Byte Buffer
FNLEN	00B7	183	Length Of Current File Name
LA	00B8	184	Current Logical File Number
SA	00B9	185	Current Secondary Address
FA	00BA	186	Current Device Number
FNADR	00BB–00BC	187–188	Pointer: Current File Name
ROPRTY	00BD	189	RS-232 Out Parity Cassette Temp
FSBLK	00BE	190	Cassette Read/Write Block Count
MYCH	00BF	191	Serial Word Buffer
CAS1	00C0	192	Tape Motor Interlock
STAL	00C1–00C2	193–194	I/O Start Address
MEMUSS	00C3–00C4	195–196	Tape Load Temps
LSTX	00C5	197	Current Key Pressed: CHR$(n) 0 = No Key
NDX	00C6	198	Number Of Characters In Keyboard Buffer (Queue)
RVS	00C7	199	Flag: Print Reverse Characters—1 = Yes, 0 = No Used
INDX	00C8	200	Pointer: End Of Logical Line For INPUT

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
LXSP	00C9–00CA	201–202	Cursor X-Y Position At Start Of INPUT
SFDX	00CB	203	Flag: Print Shifted Characters
BLNSW	00CC	204	Cursor Blink Enable: 0 = Flash Cursor
BLNCT	00CD	205	Timer: Countdown To Toggle Cursor
GDBLN	00CE	206	Character Under Cursor
BLNON	00CF	207	Flag: Last Cursor Blink On/Off
CRSW	00D0	208	Flag: INPUT Or GET From Keyboard
PNT	00D1–00D2	209–210	Pointer: Current Screen Line Address
PNTR	00D3	211	Cursor Column On Current Line
QTSW	00D4	212	Flag: Editor In Quote Mode, $00 = NO
LNMX	00D5	213	Physical Screen Line Length
TBLX	00D6	214	Current Cursor Physical Line Number
	00D7	215	Temporary Data Area
INSRT	00D8	216	Flag: Insert Mode, >0 = # INSTs
LDTB1	00D9–00F2	217–242	Screen Line Link Table Editor Temps
USER	00F3–00F4	243–244	Pointer: Current Screen Colour RAM Location
KEYTAB	00F5–00F6	245–246	Vector: Keyboard Decode Table
RIBUF	00F7–00F8	247–248	RS-232 Input Buffer Pointer
ROBUF	00F9–00FA	249–250	RS-232 Output Buffer Pointer
FREKZP	00FB–00FE	251–254	Free 0-Page Space For User Programs
BASZPT	00FF	255	BASIC Temp Data Area
	0100–01FF	256–511	Microprocessor System Stack Area
	0100–010A	256–266	Floating To String Work Area
BAD	0100–013E	256–318	Tape Input Error Log
BUF	0200–0258	512–600	System INPUT Buffer
LAT	0259–0262	601–610	KERNAL Table: Active Logical File Numbers
FAT	0263–026C	611–620	KERNAL Table: Device Number For Each File

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
SAT	026D–0276	621–630	KERNAL Table: Second Address Each File
KEYD	0277–0280	631–640	Keyboard Buffer Queue (FIFO)
MEMSTR	0281–0282	641–642	Pointer: Bottom Of Memory For Operating System
MEMSIZ	0283–0284	643–644	Pointer: Top of Memory For Operating System
TIMOUT	0285	645	Flag: Kernal Variable For IEEE Timeout
COLOR	0286	646	Current Character Colour Code
GDCOL	0287	647	Background Colour Under Cursor
HIBASE	0288	648	Top Of Screen Memory (Page)
XMAX	0289	649	Size Of Keyboard Buffer
RPTFLG	028A	650	Flag: REPEAT Key Used, $80 = Repeat
KOUNT	028B	651	Repeat Speed Counter
DELAY	028C	652	Repeat Delay Counter
SHFLAG	028D	653	Flag: Keyboard Shift Key/ CTRL Key/C = Key
LSTSHF	028E	654	Last Keyboard Shift Pattern
KEYLOG	028F–0290	655–656	Vector: Keyboard Table Set-up
MODE	0291	657	Flag: $00 = Disable Shift Keys, $80 = Enable Shift Keys
AUTODN	0292	658	Flag: Auto Scroll Down, 0 = ON
M51CTR	0293	659	RS232: 6551 Control Register Image
M51CDR	0294	660	RS232: 6551 Command Register Image
M51AJB	0295–0296	661–662	RS232 Non-Standard BPS (Time/2-100) USA
RSSTAT	0297	663	RS232: 6551 Status Register Image
BITNUM	0298	664	RS232 Number of Bits Left To Send
BAUDOF	0299–029A	665–666	RS232 Baud Rate: Full Bit Time (μs)
RIDBE	029B	667	RS232 Index To End Of Input Buffer
RIDBS	029C	668	RS232 Start Of Input Buffer (Page)

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
RODBS	029D	669	RS232 Start Of Output Buffer (Page)
RODBE	029E	670	RS232 Index To End Of Output Buffer
IRQTMP	029F-02AO	671–672	Holds IRQ Vector During Tape I/O
ENABL	02A1	673	RS232 Enables
	02A2	674	TOD Sense During Cassette I/O
	02A3	675	Temporary Storage For Cassette Read
	02A4	676	Temporary D1IRQ Indicator For Cassette Read
	02A5	677	Temp For Line Index
	02A6	678	PAL/NTSC Flag, 0 = NTSC, 1 = PAL
	02A7–02FF	679–767	Unused
IERROR	0300–0301	768–769	Vector: Print BASIC Error Message
IMAIN	0302–0303	770–771	Vector: BASIC Warm Start
ICRNCH	0304–0305	772–773	Vector: Tokenize BASIC Text
IQPLOP	0306–0307	774–775	Vector: BASIC Text LIST
IGONE	0308–0309	776–777	Vector: BASIC Character Dispatch
IEVAL	030A–030B	778–779	Vector: BASIC Token Evaluation
SAREG	030C	780	Storage for 6502 A Register
SXREG	030D	781	Storage for 6502 X Register
SYREG	030E	782	Storage For 6502 Y Register
SPREG	030F	783	Storage For 6502 SP Register
USRPOK	0310	784	USR Function Jump Instruction (4C)
USRADD	0311–0312	785–786	USR Address Low Byte/ High Byte
	0313	787	Unused
CINV	0314–0315	788–789	Vector: Hardware IRQ Interrupt
CBINV	0316–0317	790–791	Vector: BRK Instruction Interrupt
NMINV	0318–0319	792–793	Vector: Non-Maskable Interrupt
IOPEN	031A–031B	794–795	KERNAL OPEN Routine Vector

ETICHETTA	INDIRIZZO ESADECIM.	LOCAZIONE DECIMALE	DESCRIZIONE
ICLOSE	031C–031D	796–797	KERNAL CLOSE Routine Vector
ICHKIN	031E–031F	798–799	KERNAL CHKIN Routine Vector
ICKOUT	0320–0321	800–801	KERNAL CHKOUT Routine Vector
ICLRCH	0322–0323	802–803	KERNAL CLRCHN Routine Vector
IBASIN	0324–0325	804–805	KERNAL CHRIN Routine Vector
IBSOUT	0326–0327	806–807	KERNAL CHROUT Routine Vector
ISTOP	0328–0329	808–809	KERNAL STOP Routine Vector
IGETIN	032A–032B	810–811	KERNAL GETIN Routine Vector
ICLALL	032C–032D	812–813	KERNAL CLALL Routine Vector
USRCMD	032E–032F	814–815	User-Defined Vector
ILOAD	0330–0331	816–817	KERNAL LOAD Routine
ISAVE	0332–0333	818–819	KERNAL SAVE Routine Vector
	0334–033B	820–827	Unused
TBUFFR	033C–03FB	828–1019	Tape I O Buffer
	03FC–03FF	1020–1023	Unused
VICSCN	0400–07FF	1024–2047	1024 Byte Screen Memory Area
	0400–07E7	1024–2023	Video Matrix: 25 Lines × 40 Columns
	07F8–07FF	2040–2047	Sprite Data Pointers
	0800–9FFF	2048–40959	Normal BASIC Program Space
	8000–9FFF	32768–40959	VSP Cartridge ROM—8192 Bytes
	A000–BFFF	40960–49151	BASIC ROM—8192 Bytes (or 8K RAM)
	C000–CFFF	49152–53247	RAM—4096 Bytes
	D000–DFFF	53248–57343	Input Output Devices And Colour RAM Or Character Generator ROM Or RAM—4096 Bytes
	E000–FFFF	57344–65535	KERNAL ROM—8192 Bytes (Or 8K RAM)

TRADUZIONE DELLE DESCRIZIONI

* 0000: Registro direzione dati del 6510.
* 0001: Registro di input/output del 6510.
* 0002: Non usato.
* 0003-0004: Vettore di salto: conversione reale/intero.
* 0005-0006: Vettore di salto: conversione intero/reale.
* 0007: Carattere di ricerca.
* 0008: Flag: cerca gli apici alla fine di una stringa.
* 0009: Colonna di schermo dopo l'ultimo TAB.
* 000A: Flag: 0=LOAD 1=VERIFY.
* 000B: Puntatore buffer di input numero indici.
* 000C: Flag: dimensione iniziale di un array.
* 000D: Tipo di dato: $FF=stringa $00=numerico.
* 000E: Tipo di dato: $80=intero $00=reale.
* 000F: Scansione istruzione DATA/apici istruzione LIST.
* 0010: Flag: riferimento indice/chiamata di funzione da parte dell'utente.
* 0011: Flag: $00=INPUT $40=GET $98=READ.
* 0012: Flag: simbolo TAN/risultato di un confronto.
* 0013: Flag: richiesta di INPUT.
* 0014-0015: Transiente: valore intero.
* 0016: Puntatore: stack/stringhe transienti.
* 0017-0018: Ultimo indirizzo stringhe transienti.
* 0019-0021: Stack stringhe transienti.
* 0022-0025: Area puntatori routines di utilita'.
* 0026-002A: Prodotto di moltiplicazione reale.
* 002B-002C: Puntatore: inizio del testo BASIC.
* 002D-002E: Puntatore: inizio variabili del BASIC.
* 002F-0030: Puntatore: inizio arrays del BASIC.
* 0031-0032: Puntatore: fine arrays del BASIC (+1).
* 0033-0034: Puntatore: base della memoria stringhe.
* 0035-0036: Puntatore: stringa routines di utilita'.
* 0037-0038: Puntatore: indirizzo piu' alto del BASIC.
* 0039-003A: Numero di linea corrente del BASIC.
* 003B-003C: Numero di linea precedente del BASIC.
* 003D-003E: Puntatore: istruzione BASIC per CONT.
* 003F-0040: Numero di linea DATA corrente.
* 0041-0042: Puntatore: indirizzo elemento corrente di una istruzione DATA.

* 0043-0044: Vettore: routine di INPUT.
* 0045-0046: Nome variabile corrente del BASIC.
* 0047-0048: Puntatore: dato variabile corrente in BASIC.
* 0049-004A: Puntatore: variabile indice per il ciclo FOR/NEXT.
* 004B-0060: Area puntatore/dati transiente.
* 0061: Accumulatore reale #1: esponente.
* 0062-0065: Accumulatore reale #1: mantissa.
* 0066: Accumulatore reale #1: segno.
* 0067: Puntatore: costante di valutazione della serie.
* 0068: Accumulatore reale #1: cifra di overflow.
* 0069:Accumulatore reale #2: esponente.
* 006A-006D: Accumulatore reale #2: mantissa.
* 006E: Accumulatore reale #2: segno.
* 006F: Risultato di confronto del segno: accumulatore #1 contro accumulatore #2.
* 0070: Accumulatore reale #1: arrotondamento.
* 0071-0072: Puntatore: buffer cassetta.
* 0073-008A: Subroutine: preleva il prossimo byte del testo BASIC.
* 0079: Ingresso per un nuovo prelievo dello stesso byte di testo.
* 007A-007B: Puntatore: byte corrente del testo BASIC.
* 008B-008F: Valore reale d'origine della funzione RND.
* 0090: Byte immagine del registro di stato: ST.
* 0091: Flag: tasto STOP/tasto RVS.
* 0092: Costante di misura del tempo per nastro.
* 0093: Flag: 0=LOAD 1=VERIFY.
* 0094: Flag: bus seriale-carattere output bufferizzato.
* 0095: Carattere bufferizzato per bus seriale.
* 0096: Numero di sincronismo cassetta.
* 0097: Area dati transiente.
* 0098: Numero files aperti/indice della tabella files.
* 0099: Dispositivo iniziale di input (0).
* 009A: Dispositivo iniziale di output (CMD=3).
* 009B: Parita' carattere nastro.
* 009C: Flag: ricevuto byte da nastro.
* 009D: Flag: $80=modo diretto $00=modo programma.
* 009E: Registro errore passo 1 del nastro.
* 009F: Registro errore passo 2 del nastro.
* 00A0-00A2: Clock in tempo reale (approssimato) ad un sessantesimo di secondo (Jiffy).
* 00A3-00A4: Area dati transiente.
* 00A5: Contatore a ritroso di sincronizzazione cassetta.
* 00A6: Puntatore: buffer di I/O del nastro.
* 00A7: Bit di input dell'RS-232/cassetta transiente.

* 00A8: Contatore bit di input dell'RS-232/cassetta transiente.
* 00A9: Flag RS-232: controllo del bit di partenza.
* 00AA: Buffer del byte di input dell'RS-232/cassetta transiente.
* 00AB: Parita' input dell'RS-232/contatore corto della cassetta.
* 00AC-00AD: Puntatore: buffer nastro/scrolling schermo.
* 00AE-00AF: Indirizzi di fine nastro e fine programma.
* 00B0-00B1: Costanti di misura del tempo del nastro.
* 00B2-00B3: Puntatore: inizio buffer del nastro.
* 00B4: Contatore bit di output dell'RS-232/cassetta transiente.
* 00B5: Prossimo bit da inviare dell'RS-232/indicatore di fine nastro.
* 00B6: Buffer del byte di output dell'RS-232.
* 00B7: Lunghezza del nome del file corrente.
* 00B8: Numero file logico corrente.
* 00B9: Indirizzo secondario corrente.
* 00BA: Numero della periferica corrente.
* 00BB-00BC: Puntatore: nome del file corrente.
* 00BD: Parita' output dell'RS-232/cassetta transiente.
* 00BE: Contatore blocco READ/WRITE cassetta.
* 00BF: Buffer parola seriale.
* 00C0: Arresto motore del nastro.
* 00C1-00C2: Indirizzo di partenza dell'I/O.
* 00C3-00C4: Carico nastro transiente.
* 00C5: Tasto attualmente premuto [CHR$(n)]; n=0: nessun tasto premuto.
* 00C6: Numero caratteri nel buffer della tastiera.
* 00C7: Flag: stampa caratteri in reverse (1=si 0=no).
* 00C8: Puntatore: fine linea logica per INPUT.
* 00C9-00CA: Posizione (X,Y) del cursore all'inizio di un'istruzione INPUT.
* 00CB: Flag: stampa caratteri shiftati.
* 00CC: Abilitatore del lampeggio cursore: 0=lampeggio.
* 00CD: Timer: conto alla rovescia per cursore bistabile.
* 00CE: Carattere sotto il cursore.
* 00CF: Flag: ultima impostazione del cursore (lampeggiante o fisso).
* 00D0: Flag: INPUT o GET da tastiera.
* 00D1-00D2: Puntatore: indirizzo della linea di schermo corrente.
* 00D3: Colonna del cursore sulla linea corrente.
* 00D4: Flag: editing fra apici ($00=no).
* 00D5: Lunghezza fisica di una linea di schermo.

* 00D6: Numero linea fisica attuale del cursore.
* 00D7: Area dati transiente.
* 00D8: Flag: modo inserimento (>0=#INST).
* 00D9-00F2: Tavola collegamenti della linea dello schermo/editor transiente.
* 00F3-00F4: Puntatore: locazione corrente della memoria colore dello schermo.
* 00F5-00F6: Vettore: tavola di decodifica tastiera.
* 00F7-00F8: Puntatore al buffer di input dell'RS-232.
* 00F9-00FA: Puntatore al buffer di output dell'RS-232.
* 00FB-00FE: Bytes liberi della pagina zero per programmi dell'utente.
* 00FF: Area dati transiente del BASIC.
* 0100-01FF: Area stack del microprocessore 6510.
* 0100-010A: Fluttuante per area di lavoro stringhe.
* 0100-013E: Registro errori di input del nastro.
* 0200-0258: Buffer di INPUT del sistema.
* 0259-0262: Tabella del sistema operativo: numero files logici attivi.
* 0263-026C: Tabella del sistema operativo: numero periferica per ogni file.
* 026D-0276: Tabella del sistema operativo: indirizzo secondario di ogni file.
* 0277-0280: Buffer della tastiera (F.I.F.O.).
* 0281-0282: Puntatore: base della memoria per il sistema operativo.
* 0283-0284: Puntatore: top della memoria per il sistema operativo.
* 0285: Flag: variabile del sistema operativo per superamento tempo del bus IEEE.
* 0286: Codice colore del carattere corrente.
* 0287: Colore dello sfondo sotto il cursore.
* 0288: Top della memoria di schermo (pagina).
* 0289: Ampiezza del buffer di tastiera.
* 028A: Flag: ripete il tasto premuto ($80=ripete).
* 028B: Ripete il contatore di velocita'.
* 028C: Ripete il contatore di ritardo.
* 028D: Flag: tasto SHIFT/tasto CTRL/tasto C=.
* 028E: Ultima configurazione ottenuta con SHIFT.
* 028F-0290: Vettore: preparazione tabella di tastiera.
* 0291: Flag: $00=disabilita SHIFT $80=abilita SHIFT.
* 0292: Flag: scrolling automatico verso il basso (0=on).
* 0293: RS-232: immagine registro di controllo del 6551.
* 0294: RS-232: immagine registro di comando del 6551.
* 0295-0296: RS-2432: BPS non standardizzato.
* 0297: RS-232: immagine registro di stato del 6551.

* 0298: Numero di bits dell'RS-232 rimasti da inviare.
* 0299-029A: Baud-rate dell'RS-232: tempo per un bit completo (in nanosecondi).
* 029B: indice RS-232 per termine buffer di input.
* 029C: Inizio del buffer di input dell'RS-232 (pagina).
* 029D: Inizio del buffer di output dell'RS-232 (pagina).
* 029E: indice RS-232 per termine buffer di output.
* 029F-02A0: Contiene il vettore di IRQ durante l'input/output del nastro.
* 02A1: Abilita l'RS-232.
* 02A2: Lettura di TOD durante l'input/output del nastro.
* 02A3: Memorizzazione transiente per lettura cassetta.
* 02A4: Flag: D1IRQ transiente per lettura cassetta.
* 02A5: Transiente per indice di linea.
* 02A6: Flag: PAL/NTSC (0=NTSC 1=PAL).
* 02A7-02FF: Non usati.
* 0300-0301: Vettore: stampa i messaggi di errore BASIC.
* 0302-0303: Vettore: warm-start del BASIC.
* 0304-0305: Vettore: tokenizzazione del testo BASIC.
* 0306-0307: Vettore: LIST del testo BASIC.
* 0308-0309: Vettore: invio caratteri BASIC.
* 030A-030B: Vettore: valutazione dei tokens.
* 030C: Memorizzazione dell'accumulatore del 6510.
* 030D: Memorizzazione del registro X.
* 030E: Memorizzazione del registro Y.
* 030F: Memorizzazione del registro di stack pointer.
* 0310: Istruzione di salto (JMP) della funzione USR.
* 0311-0312: Indirizzo di salto della funzione USR.
* 0313: Non usato.
* 0314-0315: Vettore: interrupt hardware IRQ.
* 0316-0317: Vettore: interruzione istruzione BRK.
* 0318-0319: Vettore: interrupt non mascherabile (NMI).
* 031A-031B: Vettore: routine OPEN.
* 031C-031D: Vettore: routine CLOSE.
* 031E-031F: Vettore: routine CHKIN.
* 0320-0321: Vettore: routine CHKOUT.
* 0322-0323: Vettore: routine CLRCHN.
* 0324-0325: Vettore: routine CHRIN.
* 0326-0327: Vettore: routine CHROUT.
* 0328-0329: Vettore: routine STOP.
* 032A-032B: Vettore: routine GETIN.
* 032C-032D: Vettore: routine CLALL.
* 032E-032F: Vettore definibile dall'utente.
* 0330-0331: Vettore: routine LOAD.
* 0332-0333: Vettore: routine SAVE.
* 0334-033B: Non usati.

* 033C-03FB: Buffer di input/output del nastro.
* 03FC-03FF: Non usati.
* 0400-07FF: Area della memoria di schermo (1024 bytes).
* 0400-07E7: Matrice del video (25 linee * 40 colonne).
* 07E8-07FF: Puntatori degli sprites.
* 0800-9FFF: Area normale per programmi BASIC.
* 8000-9FFF: Area per cartuccia ROM (8192 bytes).
* A000-BFFF: Interprete BASIC (ROM).
* C000-CFFF: Area libera di memoria RAM (4096 bytes).
* D000-DFFF: Dispositivi di I/O e RAM colore, oppure ROM generatore caratteri, oppure RAM (4096 bytes).
* E000-FFFF: Sistema operativo KERNAL (ROM).

APPENDICE 10

Routine del sistema operativo

Presentiamo qui un elenco di routine in ROM usate in parte dal Kernal e in parte dal BASIC. Per quelle piu' usate abbiamo fatto una breve descrizione delle loro funzioni e del loro uso. Per usufruire di queste routine,attivate le variabili richieste, chiamate ogni routine preparatoria e poi controllate gli eventuali errori riscontrati.

Nome, Scopo
Indirizzo: in esadecimale
Registri di comunicazione: registri usati per passare o ricevere informazioni da routine del Kernal.
Routine preparatorie: routine che bisogna chiamare prima di quella desiderata.
Errori possibili: se capita un errore, viene attivato il flag di carry e il codice dell'errore e' nell'accumulatore.
Stack: numero di byte di stack usati dalla routine.
Registri usati: elenco di tutti i registri usati dalle routine del Kernal.

1. Nome: ACPTR
   Scopo: Prende dati dal bus seriale
   Indirizzo: $FFA5
   Registri di comunicazione: A; i dati ritornano in accumulatore

   Routine prep.: TALK, TKSA
   Errori possibili: vedere READST
   STACK: 13
   Registri usati: X, A

2. Nome: CHKIN
   Scopo: Apre un canale per l'input
   Indirizzo: $FFC6
   Registri di comunicazione: X; carica in X il numero di file logico da usare

   Routine prep.: OPEN
   Errori possibili: 3, 5, 6
   STACK: 0
   Registri usati: A, X

3. **Nome**: CHKOUT
   Scopo: Apre un canale per l'output
   Indirizzo: $FFC9
   Registri di comunicazione: X; carica in X il numero di file logico da usare
   Routine prep.: OPEN
   Errori possibili: 3, 5, 7
   STACK: 0
   Registri usati: A, X

4. **Nome**: CHRIN
   Scopo: Prende un carattere dal canale di input
   Indirizzo: $FFCF
   Registri di comunicazione: A; il byte dati ritorna in A
   Routine prep.: OPEN, CHKIN
   Errori possibili: vedere READST
   STACK: 0
   Registri usati: A, X

5. **Nome**: CHROUT
   Scopo: Output di un carattere
   Indirizzo: $FFD2
   Registri di comunicazione: A; carica il byte in uscita in A
   Routine prep.: OPEN, CHKOUT
   Errori possibili: vedere READST
   STACK: 0
   Registri usati: A

6. **Nome**: CIOUT
   Scopo: Trasmette un byte sul bus seriale
   Indirizzo: $FFA8
   Registri di comunicazione: A; carica il byte in uscita in A
   Routine prep.: LISTEN, (SECOND se occorre un indirizzo secondario)
   Errori possibili: vedere READST
   STACK: 0
   Registri usati: A

7. **Nome**: CLALL
   Scopo: Chiude tutti i file
   Indirizzo: $FFE7
   Registri di comunicazione: nessuno
   Routine prep.: nessuna
   Errori possibili: nessuno
   STACK: 11
   Registri usati: A, X

8. Nome: CLOSE
   Scopo: Chiude un file logico

   Indirizzo: $FFC3
   Registri di comunicazione: A; carica il file logico da chiudere in A.
   Routine prep.: nessuna
   Errori possibili: nessuno
   STACK: 0
   Registri usati: A, X

9. Nome: CLRCHN
   Scopo: Pulisce i canali I/O
   Indirizzo: $FFCC
   Registri di comunicazione: nessuno
   Routine prep.: nessuna
   Errori possibili: nessuno
   STACK: 9
   Registri usati: A, X

10. Nome: GETIN
    Scopo: Prende un carattere dal buffer di tastiera
    Indirizzo: $FFE4
    Registri di comunicazione: A; il codice del carattere ritorna in A
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 0
    Registri usati: A, X

11. Nome: IOBASE
    Scopo: Definisce la pagina di memoria I/O
    Indirizzo: $FFF3
    Registri di comunicazione: X, Y; rispettivamente byte basso e byte alto della sezione di memoria mappata per le "device" di I/O, Ritornano in X e Y
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 0
    Registri usati: X, Y

12. Nome: LISTEN
    Scopo: Comanda una "device" o il bus seriale per ricevere dati
    Indirizzo: $FFB1

Registri di comunicazione: A; carica A con numeri da 4 a 1, 3 indica la "device"
Routine prep.: nessuna
Errori possibili: vedere READST
STACK: 0
Registri usati: A

13. Nome: LOAD
Scopo: Carica o verifica RAM da una "device"
Indirizzo: $FFD5
Registri di comunicazione: A; 0 per load, 1 per verify. X e Y sono i byte basso e alto dell'indirizzo per il caricamento
Routine prep.: SETLFS, SETNAM
Errori possibili: 0, 4, 5, 8, 9
STACK: 0
Registri usati: A, X, Y

14. Nome: MEMBOT
Scopo: Mette o legge l'indirizzo piu' basso della RAM
Indirizzo: $FF9C
Registri di comunicazione: Flag di carry; 1 lettura, 0 mette l'indirizzo. X e Y byte basso e alto di questo. Se il carry e' a uno, l'indirizzo ritorna in X e Y. Se il carry e' a zero l'indirizzo in X e Y viene trasferito al puntatore all'indirizzo piu' basso della RAM
Routine prep.: nessuna
Errori possibili: nessuno
STACK: 0
Registri usati: X, Y, P

15. Nome: MEMTOP
Scopo: Mette o legge l'indirizzo piu' alto della RAM
Indirizzo: $FF99
Registri di comunicazione: Carry, X e Y come MEMBOT
Routine prep.: nessuna
Errori possibili: nessuno
STACK: 2
Registri usati: X, Y, Carry

16. Nome: OPEN
    Scopo: Apre un file logico
    Indirizzo: $FFC0
    Registri di comunicazione: nessuno
    Routine prep.: SETLFS, SETNAM
    Errori possibili: 1, 2, 4, 5, 6
    STACK: 0
    Registri usati: A, X, Y

17. Nome: PLOT
    Scopo: Mette o legge la locazione del cursore
    Indirizzo: $FFF0
    Registri di comunicazione: Carry; 1 mette la locazione del cursore, 0 per lettura.
    X: numero colonne (0 - 21).
    Y: numero di righe (0 - 22)
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 2
    Registri usati: Carry, X, Y

18. Nome: RDTIM
    Scopo: Legge un valore di 3 byte dall'orologio di sistema
    Indirizzo: $FFDE
    Registri di comunicazione: A: byte piu' significativo
    X: altro biyte piu' significativo
    Y: byte meno significativo
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 2
    Registri usati: A, X, Y

19. Nome: READST
    Scopo: Legge lo status e attiva gli errori
    Indirizzo: $FFB7
    Registri di comunicazione: A: codice di errore; vedi ST del BASIC per codici e significato
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 2
    Registri usati: A

20. **Nome**: RESTOR
Scopo: Rimette il sistema e i vettori nella condizione iniziale
Indirizzo: $FF8A
Registri di comunicazione: nessuno
Routine prep.: nessuna
Errori possibili: nessuno
STACK: 2
Registri usati: A, X, Y

21. **Nome**: SAVE
Scopo: Memorizza su una "device"
Indirizzo: $FFD8
Registri di comunicazione: A; indirizzo in pagina zero che insieme al prossimo contiene l'indirizzo di partenza della zona di memoria da memorizzare. X e Y; byte basso e alto della fine della zona di memoria in questione
Routine prep.: SETLFS, SETNAM
Errori possibili: 5, 8, 9
STACK: 0
Registri usati: A, X, Y

22. **Nome**: SCNKEY
Scopo: Scandisce la tastiera
Indirizzo: $FF9F
Registri di comunicazione: nessuno
Routine prep.: nessuna
Errori possibili: nessuno
STACK: 0
Registri usati: A, X, Y

23. **Nome**: SCREEN
Scopo: Ritorna il numero di righe e colonne dello schermo
Indirizzo: $FFED
Registri di comunicazione: X; numero di colonne, Y numero di righe
Routine prep.: nessuna
Errori possibili: nessuno
STACK: 2
Registri usati: X, Y

24. Nome: SECOND
    Scopo: Indirizzo secondario per LISTEN
    Indirizzo: $FF93
    Registri di comunicazione: A; indirizzo secondario
    Routine prep.: LISTEN
    Errori possibili: vedere READST
    STACK: 0
    Registri usati: A

25. Nome: SETLFS
    Scopo: Attiva un numero di file logico, di "device" o di un indirizzo secondario
    Indirizzo: $FFBA

    Registri di comunicazione: A; numero di file logico
    X; numero "device".
    Y; comando (ind. sec.)

    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 0
    Registri usati: A, X, Y

26. Nome: SETNAM
    Scopo: Attiva il nome file
    Indirizzo: $FFBD
    Registri di comunicazione: A; lunghezza del nome del file. X e Y; byte basso e alto dell'indirizzo di partenza della zona di memoria dove il file e' memorizzato

    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 0
    Registri usati: A, X, Y

27. Nome: SETTIM
    Scopo: Attiva l'orologio di sistema
    Indirizzo: $FFDB
    Registri di comunicazione: vedere RDTIM
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 2
    Registri usati: A, X, Y

28. Nome: STOP
    Scopo: Controlla se e' stato premuto il tasto
           STOP
    Indirizzo: $FFE1
    Registri di comunicazione: flag di zero; a 1 se STOP
                               e' stato premuto
    Routine prep.: nessuna
    Errori possibili: nessuno
    STACK: 2
    Registri usati: A, X

29. Nome: TALK
    Scopo: Comanda a una "device" sul bus seriale di
           passargli dati
    Indirizzo: $FFB4
    Registri di comunicazione: A; numero di "device"
    Routine prep.: nessuna
    Errori possibili: vedere READST
    STACK: 0
    Registri usati: A

30. Nome: TKSA
    Scopo: Manda un indirizzo secondario alla "device"
           comandata a passare dati
    Indirizzo: $FF96
    Registri di comunicazione: A; indirizzo secondario
    Routine prep.: TALK
    Errori possibili: vedere READST
    STACK: 0
    Registri usati: A

31. Nome: UNLSN
    Scopo: Comanda a tutte le "device " sul bus seriale
           di non ricevere piu' dati
    Indirizzo: $FFAE
    Registri di comunicazione: nessuno
    Routine prep.: nessuna
    Errori possibili: vedere READST
    STACK: 0
    Registri usati: A

32. Nome: UNTLK
    Scopo: Manda il comando (UNTALK) di non passare piu'
           dati a tutte le "device" sul bus seriale
    Indirizzo: $FFAB

Registri di comunicazione: nessuno
Routine prep.: nessuna
Errori possibili: vedere READST
STACK: 0
Registri usati: A

33. Nome: VECTOR
Scopo: Attiva o legge i vettori RAM di sistema
Indirizzo: $FF8D
Registri di comunicazione: X e Y; indirizzo della lista dei vettori RAM di sistema. Flag di carry; se attivo, i vettori vengono letti nella lista puntata da X e Y; se non e' attivo i contenuti della lista puntata da X e Y vengono messi nei vettori della RAM
Routine prep.: nessuna
Errori possibili: vedere READST
STACK: 2
Registri usati: Carry, X, Y

Codici degli errori

Valore	Significato
0	Routine interrotta con il tasto STOP
1	Troppi file aperti
2	File gia' aperto
3	File non aperto
4	File non trovato
5	La "device" non e' presente
6	Il file non e' di tipo "input"
7	Il file non e' di tipo "output"
8	Manca il nome del file
9	Numero illegale di "device"

Kernal ROM routine

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
E097	57495	RND
E12A	57642	SYS
E156	57686	SAVE
E165	57701	LOAD
E1BE	57790	OPEN
E1C7	57799	CLOSE
E1D4	57812	Parameters for LOAD/SAVE
E20E	57870	Check for comma
E219	57881	Parameters for OPEN/CLOSE
E264	57956	COS
E26B	57963	SIN
E2B4	58036	TAN
E30E	58126	ATN
E37B	58235	Warm start
E394	58260	Initialize
E3A2	58274	CHRGET for zero page
E3BF	58292	Initialize BASIC
E453	58451	Initialize vectors
E45F	58463	Power-up message
E500	58624	Get I/O address
E505	58629	Get screen size
E50A	58634	Put/get row/column
E518	58648	Initialize I/O
E544	58692	Clear screen
E566	58726	Home cursor
E56C	58732	Set screen pointers
E5A0	58784	Set I/O defaults
E5B4	58804	Input from keyboard
E632	58930	Input from screen
E684	59012	Quote test
E691	59025	Setup screen print
E6B6	59062	Advance cursor
E6ED	59117	Retreat cursor
E701	59137	Previous line
E716	59158	Output to screen
E87C	59516	Go to next line
E891	59537	<return> key
E8CB	59592	Set colour code
E8EA	59626	Scroll screen
E965	59749	Open space on screen
E9C8	59848	Move a screen line
E9F0	59888	Interrupt
EA87	60039	Read keyboard
EB79	60281	Keyboard select vectors
EB81	60289	Keyboard 1 – unshifted

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
EBC2	60354	Keyboard 2 – shifted
EC03	60419	Keyboard 3 – 'comm'
EC44	60484	Graphics/text control
EC4F	60495	Set graphics/text mode
EC78	60536	Keyboard 4
ECB9	60601	Video chip setup
ECF9	60665	Screen in address low
ED0B	60683	Send 'talk'
ED11	60689	Send 'listen'
ED40	60736	Send to serial bus
EDB2	60850	Serial timeout
EDEF	60911	Send 'untalk'
EE03	60931	Send 'unlisten'
EE13	60947	Receive from serial bus
EE85	61061	Serial clock on
EE8E	61070	Serial clock off
EE97	61079	Serial output '1'
EEA0	61088	Serial output '0'
EEA9	61097	Get serial in
EEBB	61115	RS-232 send
EF06	61190	Send new RS-232 byte
EF3B	61243	Disable timer
EF4A	61258	Compute bit count
EF59	61273	RS-232 receive
EF7E	61310	Setup to receive
EFC4	61380	Receive parity error
EFCC	61388	Receive overflow
EFCF	61391	Receive break
EFD2	61394	Framing error
F017	61463	Send to RS-232 buffer
F04D	61517	Input from RS-232
F0A4	61604	Check serial bus idle
F13E	61758	Get . .
F157	61783	Input
F199	61849	Get . . tape/serial/RS-232
F1CA	61898	Output . .
F20E	61966	Set input device
F250	62032	Set output device
F291	62097	Close file
F30F	62223	Find file
F31F	62239	Set file values
F32F	62255	Abort all files
F333	62259	Restore default I/O
F34A	62282	Open file
F409	62473	Open RS-232

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
F49E	62622	Load program
F5AF	62895	'searching'
F5C1	62913	Print filename
F5D2	62930	'loading/verifying'
F5DD	62941	Save program
F68F	63119	Print 'saving'
F69B	63131	Bump clock
F6BC	63164	Log PIA key reading
F6DD	63197	Get time
F6E4	63204	Set time
F6ED	63213	Check stop key
F6FB	63227	Output error messages
F72C	63276	Find any tape header
F76A	63338	Write tape header
F7D0	63440	Get buffer address
F7D7	63447	Set buffer start/end pointers
F817	63511	'press play'
F82E	63534	Check tape status
F838	63544	'press record'
F841	63553	Start tape read
F864	63588	Start tape write
F8D0	63696	Check tape stop
F8E2	63714	Set read timing
F92C	63788	Read tape bits
FA60	64096	Store tape chars.
FB8E	64398	Reset pointer
FBC8	64456	Write data to tape
FBCD	64461	IRQ entry point
FC57	64599	Write tape leader
FC93	64659	Restore normal IRQ
FCB8	64696	Set IRQ vector
FCCA	64714	Kill tape motor
FCD1	64721	Check r/w pointer
FCDB	64731	Bump r/w pointer
FCE2	64738	Power reset entry
FD15	64789	Kernal reset
FD1A	64794	Kernal move
FD30	64816	Vectors
FD50	64848	Initialize system consts.
FD9B	64923	IRQ vectors
FDA3	64931	Initialize I/O
FDDD	64989	Enable timer
FDF9	65017	Save filename data
FE00	65024	Save file details
FE07	65031	Get status
FE18	65048	Flag status

FE1C	65052	Set status
FE21	65057	Set timeout
FE25	65061	Read/set top of memory
FE27	65063	Read top of memory
FE2D	65069	Set top of memory
FE34	65076	Read/set bottom of memory
FE43	65091	NMI entry
FE66	65126	Warm start
FEB6	65206	Reset IRQ & exit
FEBC	65212	Interrupt exit
FF43	65347	Fake IRQ
FF48	65352	IRQ entry
FF81	65409	Jump table
FFF6	65526	Hardware vectors

"BASIC" ROM routine

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
A000	40960	ROM control vectors
A00C	40972	Keyword Execution vectors
A052	41042	Function vectors
A080	41088	Operator vectors
A09E	41118	Keywords
A19E	41374	Error messages
A328	41768	Error message vectors
A35B	41816	Miscellaneous messages
A389	41865	Scan stack for FOR/GOSUB
A3B8	41912	Move memory
A3FB	41979	Check stack depth
A408	41992	Check memory length
A435	42037	'out of memory'
A437	42039	Error
A469	42089	BREAK
A474	42100	'ready'
A480	42112	Ready for BASIC
A49C	42140	Handle new line
A533	42291	Re-chain lines
A560	42336	Receive input fine
A579	42361	Crunch tokens
A613	42515	Find BASIC line
A642	42562	NEW
A65E	42590	CLR
A68E	42638	Back up text pointer
A69C	42652	LIST
A742	42818	FOR
A7ED	42989	Execute statement
A81D	43037	RESTORE

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
A82C	43052	Break
A82F	43055	STOP
A831	43057	END
A857	43095	CONT
A871	43121	RUN
A883	43139	GOSUB
A8A0	43168	GOTO
A8D2	43218	RETURN
A8F8	43256	DATA
A906	43270	Scan for next statement
A928	43304	IF
A93B	43323	REM
A94B	43339	ON
A96B	43371	Get fixed point number
A9A5	43429	LET
AA80	43648	PRINT#
AA86	43654	CMD
AAA0	43680	PRINT
AB1E	43806	Print string from (y.a)
AB3B	43835	Print format character
AB4D	43853	Bad input routine
AB7B	43899	GET
ABA5	43941	INPUT#
ABBF	43967	INPUT
ABF9	44025	Prompt & input
AC06	44038	READ
ACFC	44284	Input error messages
AD1E	44318	NEXT
AD78	44408	Type match check
AD9E	44446	Evaluate expression
AEA8	44712	Constant — pi
AEF1	44785	Calculate brackets
AEF7	44791	')'
AEFF	44799	Comma
AF08	44808	Syntax error
AF14	44820	Check range
AF28	44840	Search for variable
AFA7	44967	Setup FN
AFE9	45033	OR
AFF0	45040	AND
B016	45078	Compare
B081	45185	DIM
B08B	45195	Locate variable
B113	45331	Check alphabetic
B11D	45341	Create variable
B194	45460	Array pointer subroutine

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
B1A5	45477	Value 32768
B1B2	45490	Float-fixed
B1D1	45521	Set up array
B248	45640	'bad subscript'
B24D	45645	'illegal quantity'
B34C	45900	Compute array size
B37D	45949	Perform [FRE]
B391	45969	Fix-float
B39E	45982	Perform [POS]
B3A6	45990	Check direct
B3B3	46003	Perform [DEF]
B3E1	46049	Check fn syntax
B3F4	46068	Perform [FN]
B465	46181	Perform [STR$]
B475	46197	Calculate string vector
B487	46215	Set up string
B4F4	46324	Make room for string
B526	46374	Garbage collection
B5BD	46525	Check salvageability
B606	46598	Collect string
B63D	46653	Concatenate
B67A	46714	Build string to memory
B6A3	46755	Discard unwanted string
B6DB	46811	Clean descriptor stack
B6EC	46828	Perform [CHR$]
B700	46848	Perform [LEFT$]
B72C	46892	Perform [RIGHT$]
B737	46903	Perform [MID$]
B761	46945	Pull string parameters
B77C	46972	Perform [LEN]
B782	46978	Exit string-mode
B78B	46987	Perform [ASC]
B79B	47003	Input byte parameter
B7AD	47021	Perform [VAL]
B7EB	47083	Parameters for POKE/WAIT
B7F7	47095	Float-fixed
B80D	47117	Perform [PEEK]
B824	47140	Perform [POKE]
B82D	46125	Perform [WAIT]
B849	47177	Add 0.5
B850	47184	Subtract – from
B853	47187	Perform [subtract]
B86A	47210	Perform [add]
B947	47431	Complement FAC#1
B97E	47486	'overflow'
B983	47491	Multiply by zero byte

Indirizzo (esa)	Indirizzo (decimale)	Funzioni della routine
B9EA	47594	Perform [LOG]
BA2B	47659	Multiply
BA59	47705	Multiply-a-bit
BA8C	47756	Memory to FAC#2
BAB7	47799	Adjust FAC#1/#2
BAD4	47828	Underflow/overflow
BAE2	47842	Multiply by 10
BAF9	47865	+ 10 in floating pt.
BAFE	47870	Divide by 10
BB12	47890	Divide
BBA2	48034	Memory to FAC#1
BBC7	48071	FAC#1 to memory
BBFC	48124	FAC#2 to FAC#1
BC0C	48140	FAC#1 to FAC#2
BC1B	48155	Round FAC#1
BC2B	48171	Get sign
BC39	48185	SGN
BC58	48216	ABS
BC5B	48219	Compare FAC#1 to mem.
BC9B	48283	Float-fixed
BCCC	48332	int
BCF3	48371	String to FAC(VAL)
BD7E	48510	Get ASCII digit
BDC2	48578	Print
BDCD	48589	Print line number
BDDD	48605	Float to ASCII
BF16	48918	Decimal constants
BF3A	48954	TI constants
BF71	49009	SQR
BF7B	49019	Power
BFB4	49076	Negative
BFED	49133	EXP

APPENDICE 11

Tavola dei codici dello schermo

Questo e' un elenco dei caratteri che vengono visualizzati quando il numero nella colonna valori viene messo nella memoria di schermo, ovviamente nel normale modo testo.
Set 1 e SET 2 indicano i due caratteri possibili; lo stesso valore in memoria puo' visualizzare due caratteri diversi a seconda del set che e' attivato. Per cambiare set bisogna premere contemporaneamente i tasti SHIFT e COMMODORE,oppure cambiare il valore del puntatore alla memoria caratteri che e' posto alla locazione $D018 (53272); il valore normale e' $15 (21) maiuscole e grafica, set 1 , $17 (23) maiuscole e minuscole .
NOTA: alcuni valori danno lo stesso carattere in entrambi i set.

CODICI DI SCHERMO

Value	Set 1	Set 2	Value	Set1	Set 2	Value	Set 1	Set 2	Value	Set 1	Set 2
0	@		32	SPACE		64			96	SPACE	
1	A	a	33	!		65		A	97		
2	B	b	34	"		66		B	98		
3	C	c	35	#		67		C	99		
4	D	d	36	$		68		D	100		
5	E	e	37	%		69		E	101		
6	F	f	38	&		70		F	102		
7	G	g	39	'		71		G	103		
8	H	h	40	(		72		H	104		
9	I	i	41	)		73		I	105		
10	J	j	42	*		74		J	106		
11	K	k	43	+		75		K	107		
12	L	l	44	'		76		L	108		
13	M	m	45	–		77		M	109		
14	N	n	46			78		N	110		
15	O	o	47	/		79		O	111		
16	P	p	48	0		80		P	112		
17	Q	q	49	1		81		Q	113		
18	R	r	50	2		82		R	114		
19	S	s	51	3		83		S	115		
20	T	t	52	4		84		T	116		
21	U	u	53	5		85		U	117		
22	V	v	54	6		86		V	118		
23	W	w	55	7		87		W	119		
24	X	x	56	8		88		X	120		
25	Y	y	57	9		89		Y	121		
26	Z	z	58	:		90		Z	122		
27		[	59	;		91			123		
28		£	60	<		92			124		
29		]	61	=		93			125		
30		↑	62	>		94			126		
31		←	63	?		95			127		

Nota della scansione: i codici da 108 a 114 sono errati

APPENDICE 12

Tasto corrente premuto

La locazione 197 contiene il valore del tasto premuto; se viene premuto piu' di un tasto, contiene il valore piu' alto

Valore	Tasto	Valore	Tasto
0	DEL	35	0
1	RETURN	36	M
2	CRSR→	37	K
3	F7	38	O
4	F1	39	N
5	F3	40	+
6	F5	41	P
7	CRSR ↓	42	L
8	3	43	–
9	W	44	.
10	A	45	:
11	4	46	@
12	Z	47	,
13	S	48	£
14	E	49	*
15	NESSUNO	50	;
16	5	51	CLEAR
17	R	52	NESSUNO
18	D	53	=
19	6	54	↑
20	C	55	/
21	F	56	1
22	T	57	←
23	X	58	NESSUNO
24	7	59	2
25	Y	60	SPACE
26	G	61	NESSUNO
27	8	62	Q
28	B	63	NESSUNO
29	H		
30	U		
31	V		
32	9		
33	I		
34	J		

APPENDICE 13

```
 100 rem ************ alpa ***********
 110 rem *   p. rosham, 12/4/1984    *
 111 rem *        e danny davis      *
 112 rem *                           *
 115 rem *     assembly language     *
 116 rem *      programming.aid      *
 118 rem *                           *
 120 rem ****************************
 150 goto9000
1000 rem
1010 rem linea processi
1020 rem
1030 f=0:fm=0:er=0
1040 for j=p1 to p2
1050 if c$(j,1)="  " then 1110
1053 co$=c$(j,1):ifc$(j,2)<>"  "then co$=
     co$+c$(j,2)
1054 ifc$(j,3)<>"  "then co$=co$+c$(j,3)
1055 gosub30000
1056 if er>0 then 1110
1061 if j<100 thenprint" ";
1062 if j<10 thenprint" ";
1069 printj;": ";
1070 if left$(c$(j,2),1)="l"then1075
1071 goto1080
1075 printc$(j,1)+" "+c$(j,2)+c$(j,3);" "
     ;:goto1090
1080 printc$(j,1);" ";c$(j,2);" ";c$(j,3)
     ;
1090 f=f+1
1095 printspc(8);di$
1100 if f=22 thengoto1120
1110 nextj
1120 return
2000 rem
```

```
2010 rem routine principale
2020 a$="":input"comando o linea(###) : "
     ;a$
2040 if left$(a$,1)>"9"goto3000
2042 print"▌";:for i=1 to36:print" ";:nex
     ti:printchr$(13);"▌";
2045 if left$(a$,1)<"0"goto2020
2050 K$="":forK=1to4
2060 ifmid$(a$,K,1)=" "goto2090
2065 ifmid$(a$,K,1)=""thena$"          ":j
     =val(K$):n=j:goto2170
2067 ifmid$(a$,K,1)>"9"or mid$(a$,K,1)<"0
     "thenprint"linea non valida":goto202
     0
2070 K$=K$+mid$(a$,K,1)
2080 nextK
2090 if K=5 or val(K$)=ze or val(K$)>ln g
     oto2020
2100 j=val(K$):n=j
2110 a$=right$(a$,(len(a$)-K))
2120 let K$=""
2130 for K=1 to len(a$)
2140 if mid$(a$,K,1)<>" "thenK$=K$+mid$(a
     $,K,1)
2150 nextK
2160 a$=K$
2162 if left$(a$,1)="1"thengoto2020
2170 fori=1to5step2
2180 K=int(i/2+1)
2190 c$(j,K)=mid$(a$,i,2)
2195 c$(j,K)=left$(c$(j,K)+"  ",2)
2200 nexti
2210 ifc$(n,oe)="  "then2250
2220 ifn<tpthentp=n
2230 ifn>bpthenbp=n
2240 goto2320
2250 if n<>bpgoto2280
2260 ifbp=1 or c$(bp,1)<>"  "goto2320
```

```
2270 bp=bp-oe:goto2260
2280 if n<>tpgoto2320
2290 ifc$(tp,oe)<>"  "thengoto2320
2300 iftp<>bpandtp<>lnthentp=tp+oe:goto22
     90
2310 tp=oe
2320 pp=n
2330 ifn<tpthenpp=tp:goto2380
2340 nu=ze
2350 ifpp=tpornu=0thengoto2380
2360 ifc$(pp,oe)<>"  "thennu=nu+oe
2370 pp=pp-oe:goto2350
2380 p1=pp:p2=pp
2385 ifc$(n,1)="  "then2020
2390 gosub1000
2391 ifer=1 thenprint"op-code illegale"
2392 ifer=2 thenprint"op-code non valido"
2393 ifer=3 thenprint"lungh. operando non
      valida"
2394 ifer=4 thenprint"operando illegale"
2400 goto2020
2590 rem ****** watch/nwatch
2600 input"quale indirizzo : ";qz$:xq$=ri
     ght$(("0000"+qz$),4)
2610 gosub15000:ifer=1then2600
2620 wq=xq:wq$=xq$:goto2020
2630 ifwa<>1then2640
2635 print"indirizzo ";wq$;"=  <prima> $"
     ;:et=peek(wq):gosub40000
2636 printright$(hb$,2)
2640 if peek(r)=0 thenprint"nessun progra
     mma in memoria":print:goto2645
2641 sysr
2645 ifwa<>1then2660
2650 print"indirizzo ";wq$;"=  <dopo > $"
     ;:et=peek(wq):gosub40000
2655 printright$(hb$,2)
2660 goto2020
```

```
2700 rem ********** dunp memoria
2710 dc$="0000"
2720 input"dump da quale indirizzo ";dm$
2730 xq$=right$((dc$+dm$),4):gosub15000:i
     f er=1 then2720
2740 dm=xq
2750 print"▒dump da indirizzo $";xq$
2755 g=dm
2760 formm=gto(g+176)step8:f$=""
2765 et=mm:gosub40000:printhb$;" : ";
2770 formw=0to7:mq(mw)=peeK(mm+mw)
2775 a=mq(mw):ifa<32ora>127ora>159thenf$=
     f$+chr$(32):goto2780
2776 f$=f$+chr$(a)
2780 h=int(mq(mw)/16):l=mq(mw)-16*h
2785 printmid$(d$,h+1,1)+mid$(d$,l+1,1);
2789 nextmw:printspc(8);f$
2790 nextmm
2795 getK$:ifK$=""then2795
2800 ifK$<>"m"theng=mm:goto2760
2810 goto2020
3000 rem
3005 ifa$="" goto2020
3010 rem ******** comandi
3020 K$=left$(a$,tw)
3030 ifK$="en"then5000
3040 ifK$="qu"thenstop
3044 ifK$="wa"thenwa=1:goto2600
3046 ifK$="nw"thenwa=0:goto2020
3050 ifK$="li"then4000
3060 ifK$="lo"then7000
3070 ifK$="me"then6000
3080 ifK$="ne"thenrun
3090 ifK$="ru"thengoto2630
3100 ifK$="sa"then8000
3110 ifK$="ch"then9150
3115 ifK$="du"then2700
3119 print"comando non valido "
```

```
3120 goto2000
4000 rem
4010 rem ******* routine list
4020 p1=tp:p2=bp
4025 iflen(a$)<5then4040
4030 n1=asc(mid$(a$,6,1))
4040 iflen(a$)>fr and n1>47 and n1<58 the
     n p1=val(mid$(a$,5,3))
4045 print"■"
4050 gosub1000
4060 goto2020
5000 rem
5010 rem ******* routine enter
5020 g=r:print"enter all'indirizzo $";:et
     =g:gosub40000:printhb$
5040 forj=tptobp
5050 ifc$(j,oe)="  "thengoto5470
5060 ifmid$(c$(j,tw),1,1)<>"l"then5380
5070 poke g,ze:pokeg+oe,ze:pokeg+tw,ze:po
     keg+tr,ze
5080 j1=val(mid$(c$(j,tw),tw,1)+c$(j,tr))
5090 ifleft$(c$(j,2),1)="l"thenprintj;" :
      ";c$(j,1)+" "+c$(j,2)+c$(j,3):goto5
     100
5095 printj;" : ";c$(j,1);" ";c$(j,2);c$(
     j,3)
5100 ifj1<zeorj1>lnthen5460
5110 jj$=c$(j,1):gosub20000:cj=jj
5120 ifleft$(c$(j,2),1)<>"l"then5125
5121 printj1;" : ";c$(j1,1)+" "+c$(j1,2)+
     c$(j1,3):goto5130
5125 printj1;" : ";c$(j1,1);" ";c$(j1,2);
     " ";c$(j1,3)
5130 if abs(cj)<>oethengoto5460
5140 dd=(j1<j)-(j1>j)
5150 ja=g:dp=ze
5160 ifj1=jthengoto5270
5170 cl=j+dd
```

```
5180 n1=ze:ifc$(cl,oe)="  "thengoto5220
5190 ifleft$(c$(cl,2),1)="l"goto5200
5192 n1=oe-(c$(cl,tw)<>"  ")-(c$(cl,tr)<>
     "  "):goto5220
5200 jj$=c$(cl,1):gosub20000:tj=jj
5210 n1=((tj=oe)*tr+(tj=-oe)*tw)*-1
5220 ifcl=j1anddd>0goto5270
5230 dp=dp+n1
5240 ifcl=j1thengoto5270
5250 cl=cl+dd
5260 goto5180
5270 ifcj=1thenja=ja+dd*dp+(dd>0)*-3:goto
     5310
5280 ifdd>zethendp=dp+2
5290 ifdp>126anddd<zethengoto5460
5300 ifdp>129anddd>zethengoto5460
5310 xq$=mid$(c$(j,1),1,2):gosub10000:v=x
     q
5820 pokeg,v:g=g+oe
5330 ifcj=oe thenpoke g,ja-int(ja/qk)*qk:
     g=g+oe:pokeg,int(ja/qk):g=g+oe:goto5
     360
5340 ifdd<zethendp=256-dp
5345 ifdp=0thendp=256
5350 dp=dp-tw:pokeg,dp:g=g+1
5360 print"ok"
5370 goto5470
5380 fori=1to5step2
5390 k=int(i/tw+oe)
5400 xq$=mid$(c$(j,k),1,2):gosub10000:v=x
     q
5410 ifer=1orv=-1thengoto5440
5420 pokeg,v
5430 g=g+oe
5440 nexti
5450 goto5470
5460 print"** errore-branch fuori del ran
     ge **"
```

```
5470 nextj
5480 goto2020
6000 co$=""
6010 rem ******* disassebler
6020 dc$="0000"
6030 input"disassemblo a partire da ";dm
     $
6035 xq$=right$((dc$+dm$),4):gosub15000:i
     f er=1 then6030
6038 dm=xq
6039 print"▒disassemblato da indirizzo $
     ";xq$
6040 g=dm:f=0:fm=0
6050 f=f+1:co$=""
6060 fori=1to3
6070 v(i)=peek(g):h=int(v(i)/16):l=v(i)-1
     6*h
6080 r$(i)=mid$(d$,h+1,1)+mid$(d$,l+1,1)
6090 g=g+1:nexti
6100 fori=1topc(v(1)+1):co$=co$+r$(i):nex
     ti
6110 gosub30000
6115 et=dm:gosub40000:printhb$;": ";
6117 fori=1topc(y)
6120 printr$(i);" ";
6130 nexti
6134 ww=15-(len(co$)+pc(y)):printspc(ww);
     di$
6138 g=(g-3)+pc(y):dm=g
6140 if f<>22thengoto6050
6150 getk$:ifk$=""then6150
6160 ifk$<>"m"thenf=ze:goto6050
6200 goto2020
7000 rem
7010 rem ******* load
7020 print"▒"
7030 print"load programma"
7035 input"dammi il nome del file";n$
```

```
7037 ifn$=""then7035
7040 open1,1,0,n$
7041 t=0:ff=0
7045 fori=1to200:cd$(i)="":j$(i)=""
7046 t=t+1
7047 get#1,i$(i)
7048 ifi$(i)=chr$(13)thenff=0:goto7058
7049 ifi$(i)=","thenff=1:goto7047
7050 if ff=1 goto 7057
7051 if i$(i)>chr$(47) andi$(i)<chr$(58)
     and ff=0 thenj$(i)=j$(i)+i$(i):goto7
     047
7054 ifi$(i)="*"then7059
7055 ifi$(i)=" "then7047
7057 cd$(i)=cd$(i)+i$(i):goto7047
7058 nexti
7059 close1
7060 fori=1tot-1
7061 x(i)=val(j$(i)):y=1
7062 forj=1to3
7063 c$(x(i),j)=mid$(cd$(i),y,2)
7066 c$(x(i),j)=left$(c$(x(i),j)+"  ",2)
7067 y=y+2
7068 nextj:nexti
7069 fori=1to200
7070 tp=i
7080 ifc$(i,1)<>"  "then7100
7090 nexti
7100 fori=200to1 step-1
7110 bp=i
7120 ifc$(i,1)<>"  "then7140
7130 nexti
7140 goto2020
8000 rem
8010 rem ******* save
8020 input"dammi il nome : ";n$
8030 ifn$=""then8020
8035 r$=","
```

```
8040 open1,1,1,n$
8050 fori=1to200
8052 ifc$(i,1)="  "then8080
8055 co$=c$(i,1)+c$(i,2)+c$(i,3)
8060 print#1,i;r$co$
8080 nexti
8090 print#1,"*":close1
8100 goto2020
9000 rem
9010 rem ******* inizializzazione
9020 ze=0:oe=1:tw=oe+oe:tr=oe+tw:fr=tw+tw
     :qK=256:mr=2020:ln=200
9030 dima$(15),j$(200),x(200)
9040 tp=ln:bp=oe:rem linea buffer
9050 dimc$(ln,tr),i$(1200)
9060 print"▒                inizializzazione
     "
9070 fori=oetoln
9080 forj=oetotr
9090 c$(i,j)="  "
9100 nextj
9120 nexti
9125 dim pc(256),ds$(256),r$(7),cd$(200),
     mq(176)
9126 fora=1to256:readpc(a),ds$(a):nexta
9130 d$="0123456789abcdef"
9150 print"▒"
9160 input"a quale indirizzo : ";xq$:xq$=
     left$(xq$+"0000",4)
9170 gosub15000:if er=1 or xq=0 then9160
9175 r=xq:poKer,0
9180 print"▒"
9185 et=r:gosub40000
9190 print"il programma inizia all'indiri
     zzo $";hb$
9191 goto2020
9198 rem tutti gli spazi nelle istruzioni
      data sono necessari
```

```
9199 data1,brk,2,"ora ($  ,x)",1,???,1,??
     ?,1,???,2,ora $,2,asl $,1,???
9200 data1,php,2,ora #$,1,asl a,1,???,1,?
     ??,3,ora $,3,asl $,1,???
9201 data2,"bpl "
9202 data2,"ora ($  ),y",1,???,1,???,1,??
     ?,2,"ora $  ,x",2,"asl $  ,x"
9203 data1,???,1,clc,3,"orc $    ,y",1,??
     ?,1,???,1,???,3,"ora $    ,x"
9204 data3,"asl $    ,x",1,???,3,jsr ,2,"
     and ($  ,x)",1,???,1,???,2,"bit $"
9205 data2,and $,2,rol $,1,???,1,plp,2,an
     d #$,1,rol a,1,???,3,"bit $"
9206 data3,and $,3,rol $,1,???,2,bmi $,2,
     "and ($  ),y"
9207 data1,???,1,???,1,???,2,"and $  ,x"
9208 data2,"rol $  ,x",1,???,1,sec,3,"and
      $    ,y",1,???,1,???,1,???
9209 data3,"and $    ,x",3,"rol $    ,x",
     1,???,1,rti,2,"eor ($  ,x)",1,???
9210 data1,???,1,???,2,eor $,2,lsr $,1,??
     ?,1,pha,2,eor #$,1,lsr a,1,???
9211 data3,jmp ,3,eor$,3,lsr $,1,???,2,"b
     vc "
9212 data2,"eor ($  ),y",1,???
9213 data1,???,1,???,2,"eor $  ,x",2,"lsr
      $  ,x",1,???,1,cli,3,"eor $    ,y"
9214 data1,???,1,???,1,???,3,"eor $    ,x
     ",3,"lsr $    ,x",1,???,1,rts
9215 data2,"adc ($  ,x)",1,???,1,???,1,??
     ?,2,adc $,2,ror $,1,???,1,pla
9313 data2,adc #$,1,ror a,1,???,3,jmp (,3
     ,adc $,3,ror $,1,???
9314 data2,bvs ,2,"adc ($  ),y"
9315 data1,???,1,???,1,???,2,"adc $  ,x",
     2,"ror $  ,x",1,???,1,"sei"
9316 data3,"adc $    ,y",1,???,1,???,1,??
     ?,3,"adc $    ,x",3,"ror $    ,x"
```

```
9317 data1,???,1,???,2,"sta ($  ,x)",1,??
     ?,1,???,2,sty $,2,sta $,2,"stx $"
9318 data1,???,1,dey,1,???,1,txa,1,???,3,
     sty $,3,sta $,3,stx $,1,???
9319 data2,bcc ,2,"sta ($  ,x)"
9320 data1,???,1,???,2,"sty $  ,x",2,"sta
      $  ,x"
9321 data2,"stx $  ,y",1,???,1,tya,3,"sta
      $    ,y"
9322 data1,txs,1,???,1,???,3,"sta $    ,x
     ",1,???,1,???,2,"ldy #$"
9323 data2,"lda ($  ,x)",2,ldx #$,1,???,2
     ,ldy $,2,lda $,2,ldx $,1,???
9324 data1,tay,2,lda #$,1,tax,1,???,3,ldy
      $,3,lda $,3,ldx $,1,???
9325 data2,bcs ,2,"lda ($  ),y",1,???,1,?
     ??,2,"ldy $  ,x",2,"lda $  ,x"
9326 data2,"ldx $  ,y",1,???,1,clv,3,"lda
      $    ,y",1,tsx,1,???,3,"ldy $    ,x
     "
9327 data3,"lda $    ,x",3,"ldx $    ,y",
     1,???,2,cpy #$,2,"cmp ($  ,x)"
9329 data1,???,1,???,2,cpy $,2,cmp $,2,de
     c $,1,???,1,iny,2,cmp #$,1,dex
9331 data1,???,3,cpy $,3,cmp $,3,dec $,1,
     ???,2,"bne ",2,"cmp ($  ),y"
9333 data1,???,1,???,1,???,2,"cmp $  ,x",
     2,"dec $  ,x",1,???,1,cld
9335 data3,"cmp $    ,y",1,???,1,???,1,??
     ?,3,"cmp $    ,x",3,"dec $    ,x"
9337 data1,???,2,cpx #$,2,"sbc ($  ,x)",1
     ,???,1,???,2,cpx $,2,"sbc $"
9339 data2,inc $,1,???,1,inx,2,sbc #$,1,n
     op,1,???,3,cpx $,3,"sbc $"
9341 data3,inc $,1,???,2,beq ,2,"sbc ($
     ),y",1,???,1,???,1,???
9343 data2,"sbc $  ,x",2,"inc $  ,x",1,??
     ?,1,sed,3,"sbc $    ,y",1,???,1,???
```

```
 9345 data1,???,3,"sbc $     ,x",3,"inc $
      ,x",1,???
10000 ifxq$=""thenxq=-1:er=1:return
10005 as=asc(left$(xq$,1))-48:ifas>22thene
      r=1:return
10006 if as<10andas>-1thengoto10010
10007 as=as-7:ifas<10thener=1:return
10010 xq=asc(right$(xq$,1))-48:ifxq>22then
      er=1:return
10016 ifxq<10andxq>-1thengoto10020
10017 xq=xq-7:ifxq<10thener=1:return
10020 xq=xq+16*as:er=0:return
15000 qq$=left$(xq$,2):qw$=right$(xq$,2)
15005 xq$=qq$:gosub10005:qq=256*xq
15007 ifer=1thenreturn
15010 xq$=qw$:gosub10005
15020 xq=xq+qq:xq$=qq$+qw$
15030 return
20000 jj=(jj$="90")+(jj$="b0")+(jj$="f0")+
      (jj$="30")+(jj$="d0")+(jj$="10")
20010 jj=(jj+(jj$="50")+(jj$="70"))-((jj$=
      "4c")+(jj$="6c")+(jj$="20"))
20020 return
30000 xq$=left$(co$,2):ifxq$="  "thendi$="
      ":return
30001 fl=0:sh=0:er=0
30002 gosub10000:y=xq+1:xq=0
30003 gosub32000
30004 ifer=2andfm=1then30011
30005 ifer>0orxq=-1thenc$(j,1)="  ":return
30010 jj$=xq$:gosub20000
30011 ifpc(y)=1thendi$=ds$(y):return
30015 di$=left$(ds$(y),5)
30020 ifjj<>0then30140
30030 ifright$(di$,1)="("orright$(di$,1)="
      #"thendi$=di$+"$"
30040 ifpc(y)=2thendi$=di$+right$(co$,2)
30050 if pc(y)=3then30090
```

```
30060 if len(ds$(y))=9thendi$=di$+right$(d
      s$(y),2)
30070 if len(ds$(y))=11thendi$=di$+right$(
      ds$(y),3)
30080 return
30090 op$=right$(co$,2)+mid$(co$,3,2)
30100 if len(ds$(y))=5thendi$=di$+op$
30110 if len(ds$(y))=10thendi$=di$+op$+rig
      ht$8ds$(y),1)
30120 if len(ds$(y))=11thendi$=di$+op$+rig
      ht$(ds$(y),2)
30130 return
30140 op$=right$(co$,2)+mid$(co$,3,2)
30150 ifmid$(co$,3,1)="l"thendi$=ds$(y)+ri
      ght$(co$,(len(co$)-2)):sh=1
30157 ifjj=1andfm=1thendi$=di$+op$
30170 ifjj=1andfm=0andsh=0andlen(ds$(y))=4
      thendi$=di$+op$
30175 ifjj=1 and fm=0 and sh=0 and len(ds$
      (y))=5 then di$=ds$(y)+op$+")"
30180 ifjj<>-1orfm<>1thenreturn
30190 xq$=right$(co$,2):gosub10000:zz=(g-3
      )+pc(y)
30200 ifxq>127thenxq=-1*(256-xq)
30210 et=zz+xq:gosub40000
30220 di$=di$+hb$:return
32000 ifer=1goto32090
32010 ifds$(y)="???"thener=2:goto32090
32020 iflen(co$)<>pc(y)*2andmid$(co$,3,1)<
      >"l"thener=3:goto32090
32030 forfi=2 to len(co$)
32040 ifmid$(co$,3,1)="l"then32080
32050 ifmid$(co$,fi,1)<chr$(48)thener=4
32060 ifmid$(co$,fi,1)>chr$(57)and mid$(co
      $,fi,1)<chr$(65)thener=4
32070 if mid$(co$,fi,1)>chr$(70)thener=4
32080 nextfi
32090 return
```

```
40000 hb$="":ifet>65535thenet=et-65536:got
      o40000
40003 forrr=3to0step-1
40005 rt=int(et/(16↑rr))
40010 et=et-rt*16↑rr:rt=(rt+48)-7*(rt>9)
40015 hb$=hb$+chr$(rt):nextrr
40020 return
```

APPENDICE 14

Tavola dei valori delle note musicali
(notazione anglosassone)

Questa appendice contiene un elenco dei valori di ogni nota musicale; la tavola comprende, oltre alla nota stessa, l'ottava, il valore decimale della frequenza, il byte alto (da assegnare nel registro alto della voce utilizzata), ed infine il byte basso (da assegnare nel registro basso della voce utilizzata).

NOTA MUSICALE		FREQUENZA OSCILLATORE		
NOTA	OTTAVA	VAL.DECIMALE	ALTO	BASSO
0	C-0	268	1	12
1	C#-0	284	1	28
2	D-0	301	1	45
3	D#-0	318	1	62
4	E-0	337	1	81
5	F-0	358	1	102
6	F#-0	379	1	123
7	G-0	401	1	145
8	G#-0	425	1	169
9	A-0	451	1	195
10	A#-0	477	1	221
11	B-0	506	1	250
16	C-1	536	2	24
17	C#-1	568	2	56
18	D-1	602	2	90
19	D#-1	637	2	125
20	E-1	675	2	163
21	F-1	716	2	204
22	F#-1	758	2	246
23	G-1	803	3	35
24	G#-1	851	3	83
25	A-1	902	3	134
26	A#-1	955	3	187
27	B-1	1012	3	244
32	C-2	1072	4	48

NOTA MUSICALE		FREQUENZA OSCILLATORE		
NOTA	OTTAVA	VAL.DECIMALE	ALTO	BASSO
33	C#–2	1136	4	112
34	D–2	1204	4	180
35	D#–2	1275	4	251
36	E–2	1351	5	71
37	F–2	1432	5	152
38	F#–2	1517	5	237
39	G–2	1607	6	71
40	G#–2	1703	6	167
41	A–2	1804	7	12
42	A#–2	1911	7	119
43	B–2	2025	7	233
48	C–3	2145	8	97
49	C#–3	2273	8	225
50	D–3	2408	9	104
51	D#–3	2551	9	247
52	E–3	2703	10	143
53	F–3	2864	11	48
54	F#–3	3034	11	218
55	G–3	3215	12	143
56	G#–3	3406	13	78
57	A–3	3608	14	24
58	A#–3	3823	14	239
59	B–3	4050	15	210
64	C–4	4291	16	195
65	C#–4	4547	17	195
66	D–4	4817	18	209
67	D#–4	5103	19	239
68	E–4	5407	21	31
69	F–4	5728	22	96
70	F#–4	6069	23	181
71	G–4	6430	25	30
72	G#–4	6812	26	156
73	A–4	7217	28	49
74	A#–4	7647	29	223
75	B–4	8101	31	165

NOTA MUSICALE		FREQUENZA OSCILLATORE		
NOTA	OTTAVA	VAL.DECIMALE	ALTO	BASSO
80	C–5	8583	33	135
81	C#–5	9094	35	134
82	C–0	9634	37	162
83	C#–0	10207	39	223
84	D–0	10814	42	62
85	F–5	11457	44	193
86	F#–5	12139	47	107
87	G–5	12860	50	60
88	G#–5	13625	53	57
89	A–5	14435	56	99
90	A#–5	15294	59	190
91	B–5	16203	63	75
96	C–6	17167	67	15
97	C#–6	18188	71	12
98	D–6	19269	75	69
99	D#–6	20415	79	191
100	E–6	21629	84	125
101	F–6	22915	89	131
102	F#–6	24278	94	214
103	G–6	25721	100	121
104	G#–6	27251	106	115
105	A–6	28871	112	199
106	A#–6	30588	119	124
107	B–6	32407	126	151
112	C–7	34334	134	30
113	C#–7	36376	142	24
114	D–7	38539	150	139
115	D#–7	40830	159	126
116	E–7	43258	168	250
117	F–7	45830	179	6
118	F#–7	48556	189	172
119	G–7	51443	200	243
120	G#–7	54502	212	230
121	A–7	57743	225	143
122	A#–7	61176	238	248
123	B–7	64814	253	46

GLOSSARIO

ASSEMBLER

E' un programma che prende il programma scritto in LINGUAGGIO ASSEMBLER (forma che e'comprensibile al programmatore, ma non al microprocessore) e lo converte in CODICE MACCHINA (forma comprensibile al microprocessore, ma che risulta complicata per il programmatore).

ASSEMBLER-CODICE

Vedere ASSEMBLER-LINGUAGGIO

ASSEMBLER - LINGUAGGIO

E' la forma con cui il programmatore scrive i suoi programmi; viene poi trasformata in CODICE MACCHINA dall'assemblatore (ASSEMBLER).

BINARIO

BASE 2. Usato da tutti i calcolatori, ogni cifra puo' avere solo due valori: 1 e 0 (acceso o spento). Il valore delle cifre dipende dalla posizione, come accade per il decimale:

145
= 1 X 100 + 4 X 10 + 5 X 1

in binario:

1011
= 1 X 8 + 0 X 4 + 1 X 2 + 1 X 1 = 11 decimale

BIT

Una cifra binaria, che puo' prendere i valori 1 o 0. Quando ne vengono messi insieme piu' di uno,formano un numero piu' grande (vedere BYTE e BINARIO)

BUFFER

Un'area di memoria usata per memorizzazioni temporanee, solitamente in relazione ad operazioni di Input/Output

BYTE

Unita' base della memoria del calcolatore. Una locazione di memoria contiene un byte di informazione. Ogni byte e' fatto di 8 bit e puo' memorizzare un numero fra 0 e 255. Questo numero puo' rappresentare un carattere, un valore numerico o parte di un'istruzione del microprocessore. Piu' byte insieme formano numeri piu' grandi.

CODICE MACCHINA

Qualche volta chiamato LINGUAGGIO MACCHINA e' la forma che descrive un programma che puo' funzionare direttamente sul microprocessore. Un programma in CODICE MACCHINA e' fatto di stringhe di numeri che possono essere caricati nel calcolatore dal programmatore in ESA o assemblati con un programma ASSEMBLER.

DECIMALE

Base 10. E' il metodo di numerazione usato normalmente.

DISASSEMBLER

E' un programma che prende un CODICE MACCHINA e lo stampa in linguaggio assembler.

DUMP

Fare DUMP della memoria significa visualizzarne i contenuti in forma numerica o carattere.

ESADECIMALE

BASE 16 (qualche volta chiamato ESA). E' usato in rapporto con la memoria, perche' e' facilmente convertibile in

binario. Un byte e' facilmente diviso in due sezioni di 4 bit binari ciascuna, capaci di memorizzare un numero da 0 a 15. Il numero viene rappresentato con una cifra ESA da 0 a 9, da A a F. Percio' un byte si puo' dividere in due numeri esadecimali. Il segno dollaro ($) davanti a una cifra significa che questa e' esadecimale.

GRAFICA

E' il modo con cui si puo' visualizzare qualunque cosa utilizzi la risoluzione a punti del calcolatore.

INDIRIZZO DI MEMORIA

Ogni cella di memoria ha un numero da 0 a 65535 che viene usato per riferirsi a una cella fra le 65536 all'interno del computer. Un numero usato con questo scopo viene detto indirizzo.

INTERRUPT (INTERRUZIONE)

E' un segnale mandato al microprocessore da una periferica o da un chip interno, per rendergli noto che qualcosa sta accadendo nel mondo esterno.

LINGUAGGIO MACCHINA

Vedere CODICE MACCHINA

LOCAZIONE DI MEMORIA

Un altro modo per dire indirizzo di memoria

MEMORIA

Caselle virtuali all'interno del calcolatore che sono usate per contenere numeri, istruzioni in L.M. e caratteri. Ogni casella contiene un byte alla volta. Il C64 ha 64K (65536) byte di memoria.

MICROPROCESSORE

L'unita' centrale dei processi e del controllo del calcolatore. Puo' essere paragonato al cervello umano. Il microprocessore controlla tutti i movimenti di dati, tutte le decisioni e tutti i calcoli all'interno del computer.

PAGINA ZERO

Un altro nome dei primi 356 byte della memoria da $0 a $255.

VETTORE

E' il nome dato ai byte che contengono l'indirizzo di routine residenti in ROM. Questi byte permettono all'utente di accedere a routine di input/output in ROM utilizzate dal sistema operativo e dal BASIC.

ISTRUZIONI PER IL CORRETTO CARICAMENTO DEI PROGRAMMI SU CASSETTA PER COMMODORE C64:

-Scrivere LOAD o LOAD "nome del programma" seguito dal tasto RETURN.
-Fare attenzione a digitare il nome del programma in modo esatto.

Pur eseguendo correttamente le istruzioni sopraelencate, e' possibile incontrare talvolta qualche difficolta' nel caricamento del programma. Nella massima parte dei casi il problema puo' essere facilmente risolto intervenendo sulla vite posta in prossimita' della testina di lettura del registratore per la regolazione dell'altezza -azimut-, tarandola per i toni piu' acuti. Non ottenendo il risultato, il che e' alquanto raro, e' consigliabile rivolgersi ad un tecnico.

PROGRAMMA CONTENUTO NELLA CASSETTA

La cassetta allegata al libro contiene il programma ALPA il cui funzionamento e' spiegato dettagliatamente nel capitolo quattro del presente libro.

GARANZIA CASSETTE SOFTWARE
Le cassette software originali -non manomesse- che presentassero eventuali difetti vanno spedite per la sostituzione a:
EDIZIONI JCE
Via dei Lavoratori 124
20092 CINISELLO BALSAMO

edizioni JCE

leader nell'elettronica

Ogni rivista JCE
è leader indiscusso nel settore specifico,
grazie alla ultra venticinquennale tradizione
di serietà editoriale

SELEZIONE
DI ELETTRONICA E MICROCOMPUTER
È l'unica rivista italiana a carattere esclusivamente applicativo. Si rivolge ai progettisti di apparecchiature professionali, industriali e consumer. Col materiale che riceve dalle grandi Case, redige rubriche di alto interesse tecnologico dai titoli "Microprocessori" - "Microcomputer" - "Dentro al componente" - "Tecnologie avanzate". La rivista offre al lettore la possibilità di richiedere la documentazione.

SPERIMENTARE
CON L'ELETTRONICA E IL COMPUTER
La rivista, nata per gli hobbisti e affermatasi come periodico dei giovani, non ha mai abbandonato questa categoria di lettori. Sensibile all'evoluzione, si è arricchita della materia computer, divenendo una delle pubblicazioni leader nell'ambito dell'informatica di consumo. Contiene, fra l'altro, le rubriche "Sinclub" e "A tutto Commodore" che hanno avuto un ruolo determinante nel primato della rivista.

EG COMPUTER
È il mensile di home e personal computer. Pubblicazione unica nel suo genere, ricca di spunti entusiasmanti. È la rivista per il pubblico eterogeneo attratto dall'informatica, che intende varcarne le soglie in modo stimolante e vivace.

CINESCOPIO
Unica rivista italiana di Service Radiotelevisivo, per riparatori e operatori tecnici. Sempre aggiornata sulle nuove tecniche, offre un sostegno tangibile al Service-man nell'acquisizione di una più completa e moderna professionalità.

MILLECANALI
È lo strumento critico che analizza e valuta obbiettivamente l'emittenza radio e televisiva indipendente, quale elemento di rilievo nel cammino storico dei mezzi di informazione. Offre un valido supporto tecnico agli operatori, mantenendo il proprio ruolo nei confronti delle trasmissioni private e delle loro implicazioni nel contesto sociale.

PUBBLICAZIONI JCE 1985

LIBRI DI INFORMATICA

DESCRIZIONE	CODICE	PREZZO UNITARIO
IL LIBRO DEL MICRODRIVE SPECTRUM	9001	L. 16.000
FORTH PER SPECTRUM	9005	L. 15.000
ALLA SCOPERTA DEL QL IL COMPUTER SINCLAIR	9050	L. 20.000
COME PROGRAMMARE IL TUO IBM PC	9200	L. 20.000
LA PRIMA VOLTA CON APPLE	9300	L. 16.000
ALLA SCOPERTA DELL'APPLE *IIc*	9301	L. 16.000
APPLE MACINTOSH: IL COMPUTER MAGICO	9350	L. 20.000
ATARI SERIE XL	9411	L. 16.000
IL 68000: PRINCIPI E PROGRAMMAZIONE	9850	L. 20.000

LIBRI DI INFORMATICA CON CASSETTA

SINCLAIR ZX SPECTRUM: ASSEMBLER E LINGUAGGIO MACCHINA	9000	L. 25.000
PROGRAMMARE IMMEDIATAMENTE LO SPECTRUM	9002	L. 25.000
CREATE GIOCHI ARCADE COL VOSTRO SPECTRUM	9003	L. 25.000
APPROFONDIRE LA CONOSCENZA DELLO SPECTRUM	9004	L. 30.000
PROGRAMMIAMO INSIEME LO SPECTRUM	9006	L. 30.000
BASIC & FORTRAN PER SPECTRUM	9007	L. 25.000
POTENZIATE IL VOSTRO SPECTRUM	9008	L. 30.000
49 GIOCHI ESPLOSIVI PER LO SPECTRUM	9009	L. 30.000
GRAFICA AVANZATA CON LO SPECTRUM	9010	L. 35.000
GRAFICA E SUONO PER IL LAVORO E IL GIOCO CON LO SPECTRUM	9011	L. 25.000
METTETE AL LAVORO IL VOSTRO VIC 20	9100	L. 25.000
IL MIO COMMODORE 64	9150	L. 25.000
COME PROGRAMMARE IL TUO COMMODORE 64	9151	L. 25.000
COMMODORE 64: I SEGRETI DEL LINGUAGGIO MACCHINA	9152	L. 30.000
SPRITES & SUONO DEL COMMODORE 64	9153	L. 30.000
SONY MSX BASIC	9400	L. 30.000
IMPARIAMO IL PASCAL SUL NOSTRO COMPUTER	9800	L. 25.000

SOFTWARE

GRAFICA PER TUTTI	SPECTRUM 48k e PLUS	J/0100-01	L. 25.000
MANUALE DI GEOMETRIA PIANA	SPECTRUM 48k e PLUS	J/0100-02	L. 25.000
MANUALE DI GEOMETRIA SOLIDA	SPECTRUM 48k e PLUS	J/0100-03	L. 25.000
TRIGONOMETRIA	SPECTRUM 48k e PLUS	J/0100-04	L. 25.000
MOSAICO	SPECTRUM 48k e PLUS	J/0101-01	L. 20.000
BATTAGLIA NAVALE	SPECTRUM 48k e PLUS	J/0101-02	L. 20.000
PUZZLE MUSICALE	SPECTRUM 48k e PLUS	J/0101-03	L. 20.000
SUPER EG	SPECTRUM 48k e PLUS	J/0101-04	L. 20.000
SPECTRUM WRITER (MICRODRIVE COMPATIBILE)	SPECTRUM 48k e PLUS	J/0102-01	L. 40.000
MASTER FILE (MICRODRIVE COMPATIBILE) VERSIONE ITAL.	SPECTRUM 48k e PLUS	J/0102-02	L. 40.000
BUSINESS GRAPHICS	SPECTRUM 48k e PLUS	J/0102-03	L. 25.000
INGEGNERIA: PROGRAMMA AD ELEMENTI FINITI	SPECTRUM 48k e PLUS	J/0104-01	L. 30.000
TOPOGRAFIA	SPECTRUM 48k e PLUS	J/0104-02	L. 30.000
CALCOLO TRAVI IPE	SPECTRUM 48k e PLUS	J/0104-03	L. 25.000
ENERGIA SOLARE	SPECTRUM 48k e PLUS	J/0104-04	L. 30.000
ALGEBRA MATRICIALE	SPECTRUM 48k e PLUS	J/0104-05	L. 30.000
STUDIO DI FUNZIONI	SPECTRUM 48k e PLUS	J/0104-06	L. 30.000

SOFTWARE

DESCRIZIONE		CODICE	PREZZO UNITARIO
EQUAZIONI PARAMETRICHE E PROBLEMI DI 2°GRADO	SPECTRUM 48k e PLUS	J/0104-07	L. 25.000
TOTIP	SPECTRUM 48k e PLUS	J/0105-01	L. 20.000
ASTROLOGIA	SPECTRUM 48k e PLUS	J/0105-02	L. 25.000
CAMPIONATO DI CALCIO	SPECTRUM 48k e PLUS	J/0105-03	L. 25.000
RACCOLTA DI QUIZ PER LA PATENTE (MICRODRIVE TRASFERIBILE)	SPECTRUM 48k e PLUS	J/0105-04	L. 25.000
GARDEN WARS	COMMODORE C64	J/0111-01	L. 20.000
ECONOMIA FAMILIARE	COMMODORE C64/DISCO	J/0112-02	L. 40.000
CHESS WARS	COMMODORE VIC 20 NON ESPANSO	J/0121-01	L. 20.000

LIBRI DI ELETTRONICA

DIGIT 1	2000	L. 7.000
CORSO DI PROGETTAZIONE DEI CIRCUITI A SEMICONDUTTORE	2002	L. 8.400
APPUNTI DI ELETTRONICA - VOL. 1	2300	L. 8.000
APPUNTI DI ELETTRONICA - VOL. 2	2301	L. 8.000
APPUNTI DI ELETTRONICA - VOL. 3	2302	L. 8.000
APPUNTI DI ELETTRONICA - VOL. 4	2303	L. 8.000
APPUNTI DI ELETTRONICA - VOL. 5	2304	L. 8.000
APPUNTI DI ELETTRONICA - VOL. 6	2305	L. 8.000
COSTRUIAMO UN VERO MICROELABORATORE ELETTRONICO	3000	L. 4.000
JUNIOR COMPUTER - VOL. 1	3001	L. 11.000
JUNIOR COMPUTER - VOL. 2	3002	L. 14.500
GUIDA ALL'ACQUISTO DEI SEMICONDUTTORI	4000	L. 6.000
TABELLA EQUIVALENZE SEMICOND. E TUBI ELETTRONICI PROFESSIONALI	6006	L. 5.000
TRANSISTOR CROSS-REFERENCE GUIDE	6007	L. 8.000
SELEZIONE DEI PROGETTI ELETTRONICI	6008	L. 9.000
300 CIRCUITI	6009	L. 12.500
THE WORLD TTL, IC DATA CROSS REFERENCE GUIDE	6010	L. 20.000
DIGIT 2	6011	L. 6.000
273 CIRCUITI	6014	L. 12.500
NUOVISSIMO MANUALE DI SOSTITUZIONE FRA TRANSISTORI	6015	L. 10.000
SISTEMI HI-FI MODULARI DA 3 A 1000W	6016	L. 6.000
100 RIPARAZIONI TV ILLUSTRATE E COMMENTATE	7000	L. 10.000
LE RADIO COMUNICAZIONI	7001	L. 7.500
PRATICA TV	7002	L. 10.500
99 RIPARAZIONI TV ILLUSTRATE E COMMENTATE	7003	L. 16.000
ALLA RICERCA DEI TESORI	8001	L. 6.000
LE LUCI PSICHEDELICHE	8002	L. 4.500
ACCESSORI ELETTRONICI PER AUTOVEICOLI	8003	L. 6.000
IL MODERNO LABORATORIO ELETTRONICO	8004	L. 6.000
LA PRATICA DELLE MISURE ELETTRONICHE	8006	L. 11.500

Giugno 1985

Questo libro permette all'utente inesperto di avvicinarsi al linguaggio macchina in modo semplice e diretto, con il supporto di molti esempi pratici. Scritto appositamente per gli utenti del Commodore 64,"L'ABC DEL LINGUAGGIO MACCHINA" offre una spiegazione completa di tutte le istruzioni in linguaggio macchina del 6502 e del 6510. Se siete stanchi delle limitazioni del BASIC e volete scrivere programmi o routine più veloci, più potenti e che occupino meno memoria, allora questo libro è per voi. Se non avete alcuna precedente esperienza di linguaggio di calcolatori, il taglio "semplice da capire", non in gergo, con cui è stato fatto il libro, vi permetterà di scoprire la potenza del linguaggio macchina del Commodore 64. Ogni capitolo include esempi specifici di applicazione del linguaggio macchina che potete provare ed usare sul vostro Commodore 64. Il libro e la cassetta che lo accompagna presentano anche un assemblatore che permette di scrivere direttamente programmi in linguaggio macchina. In "L'ABC DEL LINGUAGGIO MACCHINA" troverete tutto quello che vi serve per scrivere programmi in codice macchina e sfruttare al massimo le potenzialità del vostro Commodore 64.

ISBN 88-7708-006-X

Cod. 9155

L. 29.000

LIBRO + CASSETTA

9 788877 080066

edizioni JCE

L'ABC DEL LINGUAGGIO MACCHINA SUL COMMODORE 64

di DANNY DAVIS

91